Universidade Federal do Espírito Santo
Centro Tecnológico
Departamento de Engenharia Elétrica
Projeto de Graduação

Vitor Ribeiro Roriz

Implementação eficiente de máquinas de estados finitos para aplicações de controle

Vitória - ES

Dezembro/2015

Vitor Ribeiro Roriz

Implementação eficiente de máquinas de estados finitos para aplicações de controle

Parte Manuscrita do Projeto de Graduação do aluno **Vitor Ribeiro Roriz**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Universidade Federal do Espírito Santo

Centro Tecnológico

Departamento de Engenharia Elétrica

Projeto de Graduação

Orientador: Prof. Dr. rer. nat. Hans-Jorg A. Schneebeli

Vitória - ES

Dezembro/2015

Vitor Ribeiro Roriz
Implementação eficiente de máquinas de estados finitos para aplicações de controle/ Vitor Ribeiro Roriz. – Vitória - ES, Dezembro/2015-
84 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. rer. nat. Hans-Jorg A. Schneebeli

Tese (Graduação) – Universidade Federal do Espírito Santo
Centro Tecnológico
Departamento de Engenharia Elétrica
Projeto de Graduação, Dezembro/2015.

1. Máquinas de estados finitos. 2. Diagramas de decisão binária. 2. Sistemas embarcados. I. Orientador. II. Universidade Federal do Espírito Santo. III. Departamento de Engenharia Elétrica. IV. Título

Vitor Ribeiro Roriz

Implementação eficiente de máquinas de estados finitos para aplicações de controle

Parte Manuscrita do Projeto de Graduação do aluno **Vitor Ribeiro Roriz**, apresentado ao Departamento de Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Engenheiro Eletricista.

Comissão Examinadora:

Prof. Dr. rer. nat. Hans-Jorg A. Schneebeli
Orientador

Prof. Dra. Eliete Maria de Oliveira Caldeira
Avaliadora

Eng. Philipe Rangel Demuth
Avaliador

Vitória - ES
Dezembro/2015

Este trabalho é dedicado à querida Vovó Zezé que me ensinou coisas que livros não registram e que, mesmo sem olhos vivos, olha e guarda por mim.

Agradecimentos

Agradeço muitíssimo a meus pais e avós, que nunca me cobraram nada mas comemoraram cada pequena conquista minha como um Nobel. Às minhas irmãs, pelo amor e exercícios de paciência. A toda minha família. Agradeço aos mestres pelos desafios impostos. Em especial a meu orientador, Prof. Hans, por nos permitir testemunhar conhecimentos de grandeza rara, alinhados à vontade de ensinar e à paciência. À Prof. Eliete e ao Mestre e amigo Demuth, pela gentileza de terem aceitado compor minha banca de avaliação mesmo diante da escacez de tempo. Aos Professores Raquel, Mattedi e Munaro, pelos ensinamentos e oportunidades. A todos meus amigos do PET (PND), do Nacional, da 10/1 (em especial ao amigo Fernando Martinelli, pelo recorde de maior quantidade de trabalhos em equipe sem mortes), da Elétrica, da UFES e da vida. Obrigado amigos, por muitas vezes atrapalharem a realização deste e outros trabalhos e assim contribuírem na manutenção da minha sanidade. Agradeço a Deus, seja lá o que ele, ou ela, for exatamente. Muito obrigado por me permitir conhecer e caminhar com todas estas pessoas.

"O bem que praticares, em algum lugar, é teu advogado em toda parte."
(Francisco Cândido Xavier)

Resumo

Este trabalho narra o processo de desenvolvimento de um conjunto de software capaz de representar, de maneira eficiente, uma máquina de estados finitos qualquer a partir de um arquivo descritor, a fim de servir de base para a implementação de software de um PLC (Controlador Lógico Programável, do inglês *Programmable Logic Controller*). Dois programas foram desenvolvidos. O primeiro programa criado absorve a lógica da máquina de estados descrita e gera uma representação para a máquina em diagramas de decisão binária (BDD), sendo esta uma representação mínima, ordenada e canônica. Essa descrição é exportada em dois formatos diferentes (C e XML). O primeiro formato pode ser facilmente importado por qualquer programa escrito em C, mas assume o compromisso de descrever a máquina de estados em tempo de compilação. O segundo formato, XML, traz a flexibilidade de poder ser utilizado por software de qualquer linguagem, com o tratamento devido. O formato garante ainda a possibilidade de alteração da descrição do sistema em tempo de execução. O segundo programa desenvolvido pode ser embarcado em microcontroladores e computadores que contem com compilador para a linguagem C. Ele é capaz de importar as descrições lógicas nos dois formatos enunciados. A partir dos BDD importados, o software pode ainda, de maneira eficiente (pela própria natureza da representação em BDD), estimar todas as saídas do sistema assim como seu próximo estado, baseando-se no estado atual e nas entradas do sistema. Implementando portanto a máquina de estados finitos pretendida. A implementação aqui proposta evita verificações redundantes e desordenadas das entradas e estados do sistema, muito presentes em implementações textuais com *if-then-else*, por exemplo.

Palavras-chave: Máquinas de estados finitos, diagramas de decisões binária, sistemas embarcados, PLC.

Abstract

This paper narrates the process of developing a set of software which is capable of represent, in an efficient manner, a finite-state machine from a descriptor file, to serve as basis for the software implementation of a PLC (Programmable Logic Controller). Two programs have been developed. The first program created absorbs the logic of the state machine described and generates a representation for the machine in binary decision diagrams (BDD), which is a minimum, ordered and canonical representation. This description is exported in two different formats (C and XML). The first format (C), can be easily imported by any program written in C, but it is committed to describe the finite-state machine at compile time. The second format (XML), brings the flexibility of being used by software in any language, with proper treatment. The format also ensures the possibility of changing the system description at runtime. The second program developed can be embedded in computers and microcontrollers which contain compiler for the C language. It is able to import the logical descriptions from both C and XML files. From the imported BDD, the software can also effectively (by the very nature of BDD representation), estimate all system's outputs as well as its next state, based on the current state and the system's inputs, implementing so the desired finite-state machine. Such implementation avoids redundant and disordered verifications of the inputs and states of the system, very present in textual implementations with if-then-else, for example.

Keywords: Finite-state machines, binary decision diagrams, embedded systems, PLC.

Lista de ilustrações

Figura 1 – PLC modelo PM595 da ABB. 24
Figura 2 – Diagrama geral do problema proposto. 27
Figura 3 – Diagrama de estados da máquina de estados do exemplo proposto. . . . 30
Figura 4 – Representação textual em linguagem C da máquina de estados do problema proposto. 32
Figura 5 – BDD para uma função f de 3 variáveis. 36
Figura 6 – Propriedades necessárias a um ROBDD. 37
Figura 7 – BDD ordenado e reduzido para uma função f de 3 variáveis. 38
Figura 8 – Diagrama de estados de uma catraca simples. 39
Figura 9 – BDD de uma máquina de estados finitos. 40
Figura 10 – Diagrama geral dos programas desenvolvidos. 41
Figura 11 – Diagrama genérico do software Floresta. 42
Figura 12 – Diagrama detalhado do software Floresta. 43
Figura 13 – Tela do Software OrangeCad. 44
Figura 14 – Exemplo de um arquivo descritor no formato ESPRESSO. 45
Figura 15 – Descrição do sistema em diagramas BDD. 47
Figura 16 – Arquivo descritor ESPRESSO para XOR de 3 entradas. 48
Figura 17 – Arquivo descritor ESPRESSO para XOR de 3 entradas, com marcações. 49
Figura 18 – Diagrama BDD de um sistema XOR de 3 entradas em tabela. 50
Figura 19 – Diagrama BDD de um sistema XOR de 3 entradas em árvore. 50
Figura 20 – Definição do tipo treeNode. 52
Figura 21 – Diagrama de criação de árvores treeNode. 53
Figura 22 – Exemplo de escrita feita pela função plantaArvoreC. 56
Figura 23 – Exemplo de escrita feita pela função plantaFlorestaC. 57
Figura 24 – Exemplo de arquivo XML gerado, sistema XOR. 58
Figura 25 – Exemplo de arquivo C gerado, sistema XOR. 59
Figura 26 – Dinâmica do software PLC Engine. 60
Figura 27 – Pseudocódigo de execução de PLC Engine. 60
Figura 28 – Função de atualização das saídas do sistema. 63
Figura 29 – Função de atualização do estado da máquina de estados. 64
Figura 30 – Diagramas de estados do problema proposto. 67
Figura 31 – Arquivo descritor ESPRESSO do problema proposto. 67
Figura 32 – Diagramas BDD das saídas do problema proposto. 68
Figura 33 – Arquivo .C gerado pelo software Floresta na exportação das estruturas BDD para o problema proposto. 69
Figura 34 – Gráfico em árvore do BDD da saída 1. 70

Figura 35 – Gráfico em árvore do BDD da saída 2. 71
Figura 36 – Gráfico em árvore do BDD da saída 3. 72
Figura 37 – Gráfico em árvore do BDD da saída 4. 73
Figura 38 – Gráfico em árvore do BDD da saída 5. 74
Figura 39 – Árvores BDD importadas do arquivo .C ou XML pelo software PLC Engine 75
Figura 40 – Simulação de PLC Engine para o problema proposto, parte I. 76
Figura 41 – Simulação de PLC Engine para o problema proposto, parte II. 77
Figura 42 – Simulação de PLC Engine para o problema proposto, parte III. 78

Lista de tabelas

Tabela 1 – Tabela descritiva da máquina de estados do exemplo proposto. 33
Tabela 2 – Tabela compacta descritiva da máquina de estados do exemplo proposto. 34
Tabela 3 – Representação tabular da máquina de estados de uma catraca simples. 39
Tabela 4 – Recomendação de parametrização da função bdd_init. 47

Lista de abreviaturas e siglas

BDD	*Binary Decision Diagrams*
FSM	*Finite-state Machines*
IDE	*Integrated Development Environment*
PLC	*Programmable Logic Controller*
ROBDD	*Reduced Ordered Binary Decision Diagrams*
PID	Proporcional Integral Derivativo
PLC	*Programmable Logic Controller*
SD	*Secure Digital*
SO	Sistema Operacional
VHDL	*Very High Speed Integrated Circuits*

Sumário

1 INTRODUÇÃO . . . 23
1.1 Motivação . . . 23
1.2 Descrição da Área . . . 24
1.3 Definição do Problema . . . 25
1.4 Metodologia . . . 27
1.5 Estrutura do Trabalho . . . 28

2 MÁQUINAS DE ESTADOS FINITOS PARA CONTROLE . . . 29
2.1 Representação Gráfica . . . 30
2.2 Representação Textual . . . 31
2.3 Representação Tabular . . . 32

3 USO DE BDD PARA REPRESENTAÇÃO DE MÁQUINAS DE ESTADOS FINITOS . . . 35
3.1 Os Diagramas de Decisão Binária (BDD) . . . 35
3.2 Representando Máquina de Estados Finitos . . . 38

4 IMPLEMENTAÇÃO . . . 41
4.1 Programa Host, Floresta . . . 42
4.1.1 Arquivo de descrição do sistema . . . 44
4.1.1.1 Arquivo ESPRESSO . . . 44
4.1.2 Identificação dos parâmetros de entrada do sistema . . . 45
4.1.3 Descrição do sistema em diagramas BDD . . . 46
4.1.4 Montagem das árvores BDD . . . 51
4.1.4.1 O tipo treeNode . . . 52
4.1.4.2 Construção de árvores simples . . . 52
4.1.4.3 Construção de árvores indexadas . . . 54
4.1.5 Exportação das árvores BDD . . . 55
4.1.5.1 Exportando Floresta em um arquivo C . . . 55
4.1.5.2 Exportando Floresta em um arquivo XML . . . 57
4.2 Programa PLC Engine . . . 59
4.2.1 Importando Floresta a partir de um arquivo C. . . . 61
4.2.2 Importando Floresta a partir de um arquivo XML . . . 61
4.2.3 Atualizando as saídas do sistema . . . 62
4.2.4 Atualizando o estado do sistema . . . 63

5 RESULTADOS **65**
5.1 O Problema de Exemplo Proposto **65**
5.2 Resultados obtidos **66**
5.2.1 Executando Floresta 68
5.2.2 Executando PLC Engine 74

6 CONCLUSÃO **81**

REFERÊNCIAS **83**

1 Introdução

1.1 Motivação

Os PLCs , dominam hoje, e desde muito tempo, o mercado de automação industrial em todo o mundo, seja controlando processos químicos em uma plataforma de petróleo, garantindo o pleno funcionamento de caldeiras de siderúrgicas ou manipulando eventos discretos em esteiras de engarrafamento em cervejarias. Há alguns grandes fabricantes que fornecem não só PLCs de grande aceitação, bem como todo o necessário para a implementação da atividade de controle com estes dispositivos, como cartões de entrada e saída e softwares de desenvolvimento. Ainda que bem aceitos como solução para automação e tendo preços relativamente pequenos se comparados com o preço total de grandes plantas industriais, os PLCs não possuem um preço absoluto baixo e possuem muitas vezes hardware com capacidade de processamento e recursos inferiores aos de outros produtos como computadores pessoais e microcontroladores de baixo custo.

Assim, uma discussão recorrente entre os colegas de curso e os interessados nos tópicos de sistemas embarcados e automação é o porquê da não substituição de PLCs por microcontroladores. A resposta imediata seria a de que um PLC é um produto completo, feito para funcionar 24 horas por dia e 7 dias por semana, implementando basicamente um conjunto de máquinas de estados, enquanto microcontroladores são dispositivos mais genéricos, que podem ser utilizados para o desenvolvimento de diversos produtos, inclusive PLCs. O desenvolvimento de software para um sistema baseado em microcontroladores é uma tarefa mais complexa e que exige processos de desenvolvimento e verificação complexos.

O que se percebe é que as grandes empresas que produzem PLCs fidelizam seus clientes vendendo a eles cartões e software compatíveis com seus produtos, desta forma, uma vez adquirido um controlador de um certo fabricante, há uma certa inércia por parte da indústria em seguir utilizando as mesmas soluções, mesmo para aplicações mais simples. Observa-se ainda que clientes em potencial, com plantas de menor porte e que possivelmente demandariam automação para algumas atividades, como produtores agrícolas médios ou uma pequena fábrica de biscoitos, ficariam presos a tais controladores que, talvez, tenham um custo alto para eles.

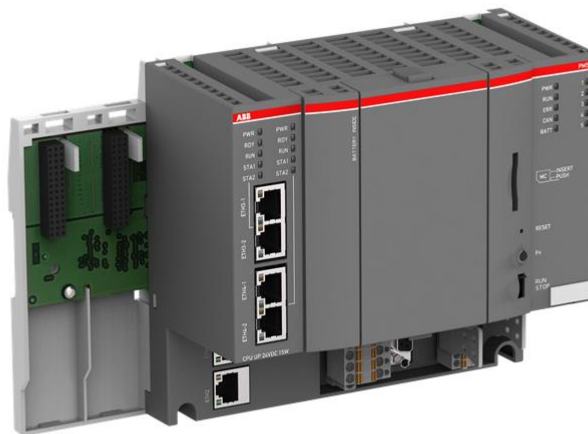
Assim a motivação para realização deste trabalho é o desejo de fomentar a elaboração de um PLC simples baseado em um microcontrolador para o controle de eventos discreto, mais especificamente o desejo de desenvolvimento do *firmware* nele embarcado, que é parte fundamental deste tipo de dispositivo. Isso porque, apesar de existir recurso su-

ficiente em um microcontrolador para operar diversos processos controlados por um PLC, é ainda necessário que seu *firmware* absorva a lógica de controle exigida pelo usuário da mesma forma que aconteceria em tal dispositivo.

1.2 Descrição da Área

Um PLC é um controlador microprocessado com múltiplas entradas e saídas, podendo incorporar periféricos como conversores analógico-digitais, blocos PID (Proporcional Integral Derivativo) entre outros. Ele é controlado por um conjunto de instruções armazenado em uma memória que descreve o seu comportamento diante da variação dos sinais de entrada. Sua função básica é ler as variáveis pertinentes do processo, por meio de sensores e transmissores enlaçados com suas entradas, e gerar saídas diretas ou através de atuadores externos que, conectados à planta, farão com que determinado processo seja controlado de acordo com uma lógica requerida. Um exemplo de PLC é o PM595 da ABB, exibido na Figura 1.

Figura 1: PLC modelo PM595 da ABB.



Fonte: www.abb.com (2015)

Segundo Bliesener, Ebel e Löffler (2002, p. 9), o primeiro PLC foi desenvolvido por um grupo de engenheiros da *General Motors* em 1968 quando empresas estavam procurando por uma alternativa para substituir sistemas de controle complexos baseados em relés. Buscava-se um novo sistema com programação simples, mudanças na lógica diretora sem necessidade de refazer ligações internas, menor tamanho físico do dispositivo e menores preços de sistema e de manutenção.

O desenvolvimento subsequente trouxe um sistema que possibilita a simples conexão de sinais binários, que se “ligam” de acordo com as ordens do programa de controle escrito. Desde então, este sistema simples tem evoluído e agregado novas funcionalidades, antes inimagináveis, como visualização dos processos associados, processamento de sinais, funções de contadores e temporizadores e aplicação de técnicas de controle.

De uma maneira mais genérica pode-se dividir o PLC em três partes principais:

1. O hardware, que corresponde ao microprocessador, seus periféricos (periféricos de comunicação, temporizadores e etc.) e suas unidades de memória.
2. Cartões de Entrada/Saída, que geralmente podem ser adicionados externamente, fazendo a expansão do hardware até certo limite e permitem a adequação do sistema para o recebimento dos sinais de entradas, vindos de diversos sensores e também o fornecimento de sinais de saídas, levados a atuadores pertinentes.
3. Software/firmware, é a “alma” do sistema, código residente na memória não volátil do PLC, faz a gerência de todos os recursos disponíveis, fazendo-os funcionar de acordo com a lógica requerida pelo usuário, a qual deve ser capaz de absorver.

Com a evolução da indústria de microeletrônica, há atualmente microcontroladores comerciais com recursos mais que suficientes para a implementação de hardware de um controlador deste tipo. Assim, o presente trabalho propõe a elaboração de software para a construção de um PLC, que seja capaz de implementar funcionalidades básicas destes sistemas e seja portável a posterior embarque em hardware com arquitetura genérica. Isso é, deseja-se desenvolver um software que implemente a lógica capaz de resolver um problema de controle, e que seria implementada por um PLC, em microcontroladores.

1.3 Definição do Problema

Deve-se compreender o que o software embarcado em um PLC faz. De forma genérica, esse software precisa ler as entradas do sistema, identificando em que estado de operação o sistema está. Em seguida, dado este conjunto de entradas em um dado instante, deve-se calcular os valores para as saídas do sistema, definindo tanto seu próximo estado de operação, quanto os valores de suas saídas físicas, o que configura a implementação de uma máquina de estados finitos, como será visto no Capítulo 2. As respostas esperadas são regidas pela lógica descrita pelo usuário, ou seja, pela relação lógica que o usário deseja que haja entre entradas e saídas do sistema.

De maneira geral, pode-se dividir o software de um PLC em dois programas diferentes. O primeiro não é propriamente embarcado no PLC mas sim hospedado por um computador à parte, utilizado para desenvolvimento. Este é o software que conversa com

o usuário e o permite descrever a planta e as ações necessárias a seu controle. O segundo é o que de fato é embarcado no dispositivo que se liga à planta e é o responsável por executar as máquinas de estados criadas pelo primeiro.

Quando deseja-se controlar uma planta qualquer através de um PLC, em suma, deve-se explicitar de alguma forma a sequência de ações que satisfaçam as condições de operação pretendidas, ou seja, a lógica que rege o processo. Isso normalmente é feito por meio de um software especializado em um computador, no qual o usuário programa tal lógica. Em seguida o software a transmite ao controlador lógico programável que, por sua vez, implementa-a. Há diversas formas de o usuário definir o conjunto de ações, seja de maneira gráfica, textual ou tabular. Deve-se notar que o objeto de estudo do projeto não é o software de descrição do processo mas sim o software que irá receber essa representação e implementá-la em um hardware qualquer que permita compilação para a linguagem de desenvolvimento a ser escolhida, seja um computador ou um microcontrolador. Para isto é necessária a verificação de uma série de funções booleanas. Como o processo pode ser muito grande e, por consequência, precisar de muitas funções e estados para descrevê-lo, estas verificações devem ocorrer de maneira eficiente, por meio de técnica adequada que será abordada no Capítulo 4, relativo à implementação.

O problema a ser resolvido é então dividido em 2 problemas menores. O problema número 1 é o de absorver a lógica proposta pelo usuário, ou seja, as relações lógicas entre todas as entradas e saídas do sistema e implementar estruturas capazes de descrevê-las de forma eficiente em um sistema embarcado. Essas estruturas deverão ser exportadas então para esse sistema.

O problema número 2 é o da criação de uma *engine*. De acordo com Wiesen (2015), uma *engine* de software é parte de um programa de computador que serve como núcleo para um fragmento maior do software. Quando usado em um contexto geral de desenvolvimento de software, uma *engine* tipicamente se refere aos elementos centrais de um programa em particular, sendo que isto geralmente não inclui características como a de interface do usuário. Ele ainda exemplifica dizendo que em um sistema operacional (SO), uma *engine* pode ser o código fonte que estabele a hierarquia de arquivos, métodos de entrada e saída e os meios pelos quais o SO se comunica com outro software ou hardware.

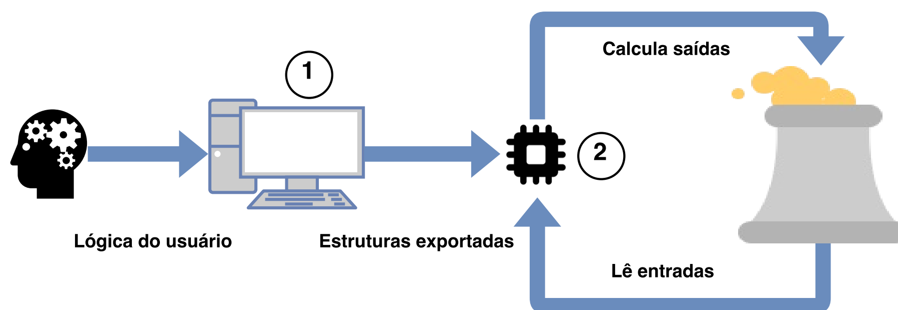
No escopo do projeto a *engine* será o fragmento de código que absorverá e utilizará as funções e estruturas desenvolvidas. Como o PLC deve responder rapidamente aos sinais de entrada, a *engine* a ser desenvolvida deve, de forma eficiente, ser capaz de importar a máquina de estados do usuário, importando as estruturas desenvolvidas no problema 1 e, baseando-se nas entradas lidas do sistema, calcular as respostas do sistema (incluindo o novo estado da máquina) utilizando as estruturas implementadas que descrevem a lógica do usuário. Isto deve ser feito de maneira automática e eficiente, isolando ao máximo o usuário da implementação para que erros humanos sejam evitados.

Assim, destacam-se os problemas que são identificados na Figura 2.

- Problema 1: Desenvolvimento de um programa que será executado em ambiente de desenvolvimento (computador genérico) e realize de forma automática a absorção da lógica do usuário e a construção de estruturas a serem embarcadas que descrevam tal lógica.

- Problema 2: Desenvolvimento de um segundo programa, (*engine*), que possa ser executado em um sistema embarcado e realize a importação das estruturas geradas e o cálculo das respostas esperadas a partir destas estruturas.

Figura 2: Diagrama geral do problema proposto.



Fonte: Produzido pelo autor.

1.4 Metodologia

No presente trabalho adotar-se-á a seguinte metodologia:

- Realização de estudo sobre a técnica a ser utilizada para representar e manipular expressões booleanas, que são ações fundamentais à construção do projeto. A técnica escolhida foi a de construção de diagramas de decisão binária (BDD).

- Definição da linguagem C (C99) para desenvolvimento do projeto. Isto se deve principalmente ao fato de C já ser uma linguagem consolidada e muito popular, o que garante disponibilidade de ampla documentação e recursos, como bibliotecas já desenvolvidas. A escolha leva em consideração também a máxima de que a linguagem deve ser compilável para a maioria dos microcontroladores presentes em sistemas embarcados.

- Realização de pesquisas para escolha da biblioteca em C que implemente as funções necessárias segundo a técnica escolhida e estudada. É fundamental que a biblioteca seja bem documentada e referenciada.
- Definição da linguagem XML como uma das linguagens de descrição utilizadas para armazenamento e exportação de dados do projeto. Também realizou-se pesquisas para a escolha de biblioteca em C que implemente funções básicas para escrita de arquivos neste formato.
- Execução de testes básicos com as funções das bibliotecas de manipulação de expressões booleanas e de descrição escolhidas. Desenvolvimento de bibliotecas próprias que suportem todos os requisitos do projeto utilizando tais funções básicas.
- Implementação dos software propostos utilizando as bibliotecas desenvolvidas.
- Execução de testes com os software desenvolvidos e resolução de problemas completos para documentação.
- Realização de novas pesquisas para revisão dos tópicos abordados a fim de se elaborar a documentação do projeto, seguida da documentação propriamente dita.

1.5 Estrutura do Trabalho

O trabalho aqui apresentado é composto por 6 capítulos que têm a pretensão de narrar seu desenvolvimento.

O Capítulo 1, introduz os tópicos que serão aqui abordados e firma os compromissos e pretensões do projeto.

O Capítulo 2 traz conceitos de máquinas de estados finitos, mostrando suas principais formas de representação.

O Capítulo 3 define o que são diagramas de decisão binária. Esse não tem a pretensão de se aprofundar matematicamente no tema, o que foge ao escopo do trabalho, mas sim de fazer o leitor entender como os diagramas são úteis e aplicáveis ao desenvolvimento do projeto.

O Capítulo 4 descreve todo o processo de implementação do projeto.

O Capítulo 5 mostra os resultados do projeto, narrando a resolução de um problema completo.

Por fim, o Capítulo 6 relata as impressões e conclusões sobre o trabalho.

2 Máquinas de Estados Finitos para Controle

Como mencionado no Capítulo 1 deste trabalho, genericamente, a função de um PLC é a de implementar a máquina de estados finitos que atue resolvendo o problema de controle proposto pelo usuário.

> Como outras teorias motivadas pelas necessidades da ciência e da engenharia, a teoria da máquinas de estados finitos diz respeito a modelos matemáticos que servem como aproximações para fenômenos físicos ou abstratos. A significância dessa teoria é a de que seus modelos não estão restritos a nenhuma área científica particular, mas são diretamente aplicados a problemas em praticamente todos os campos de pesquisa - de psicologia à administração de empresas, e de comunicações à linguística. Pode-se encontrar as ideias e técnicas da teoria de máquinas de estados finitas empregadas a problemas tão aparentemente não relacionados como a pesquisa da atividade neurológica humanas, a análise da sintaxe da língua inglesa e o projeto de computadores eletrônicos. Em uma época que a taxa de progresso científico depende fortemente da cooperação interdisciplinar, a natureza unificadora desta teoria é de valor aparente (GILL et al., 1962, p.1, tradução nossa).

Assim, na utilização deste trabalho, uma máquina de estados finitos é apresentada como uma forma de se modelar um problema de engenharia que o usuário pretenda resolver. O estado de um modelo desse tipo pode ser caracterizado pelas condições do sistema em um determinado instante. A máquina de estados então define suas saídas baseando-se tanto nas entradas do sistema como no estado atual do sistema.

> O estado de um sistema é um sumário de tudo que o sistema precisa saber sobre entradas passadas para produzir saídas. Ele é representado por uma variável de estado $s \in \Sigma$, na qual Σ é o conjunto de todos os possíveis estados para o sistema. Uma **máquina de estados finitos (FSM)** é uma máquina de estados na qual Σ é um conjunto finito. Em uma máquina de estados finitos, o comportamento de um sistema é modelado por um conjunto de estados e regras que governam as transições entre eles (PTOLEMAEUS, 2014, p. 187, tradução nossa, grifo do autor).

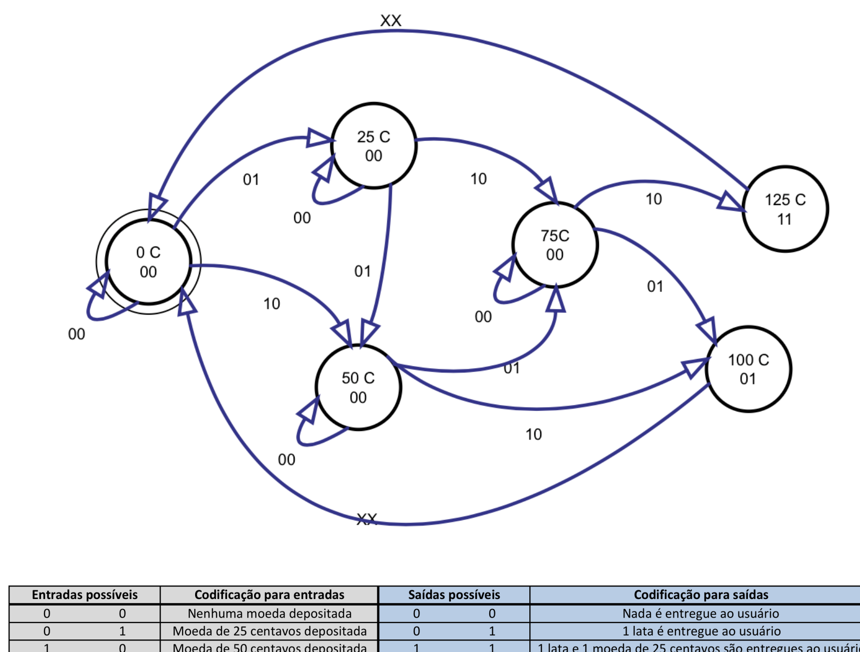
Neste contexto, deve-se definir de que forma o projetista do sistema pode definir a máquina de estados adequada a sua solução. Há diversas maneiras de representação que divergem, sobretudo, em relação a sua natureza, podendo esta ser gráfica, textual ou tabular. O Capítulo presente se restringe a analisar tais formas de representação, suficientes ao entendimento do projeto, porém destaca-se que existem muitas outras formas e técnicas que podem ser utilizadas, como é discutido por Samek (2008).

2.1 Representação Gráfica

A representação da máquina de estados na forma gráfica pode ser feita através do chamado diagrama de estados. Neles, cada estado é representado por um círculo. Cada círculo contém o nome atrelado àquele estado e, no modelo da máquina de Moore, a saída do sistema para aquele estado. Os círculos são ligados por setas que representam a transição de um estado a outro. As seta vêm acompanhadas de um texto indicativo que informa qual configuração de entrada permite a transição indicada.

Para um melhor entendimento, propomos uma máquina de estados finitos de exemplo, na forma de diagrama de estados, que pode ser vista na Figura 3. Pode-se ver o

Figura 3: Diagrama de estados da máquina de estados do exemplo proposto.



Entradas possíveis		Codificação para entradas	Saídas possíveis		Codificação para saídas
0	0	Nenhuma moeda depositada	0	0	Nada é entregue ao usuário
0	1	Moeda de 25 centavos depositada	0	1	1 lata é entregue ao usuário
1	0	Moeda de 50 centavos depositada	1	1	1 lata e 1 moeda de 25 centavos são entregues ao usuário

Fonte: Produzido pelo autor.

diagrama de estados para o problema de uma máquina "não muito inteligente"de venda de refrigerantes. Esta máquina só vende 1 tipo de refrigerante, que custa, incrivelmente, apenas R$1,00, ou 100 centavos (100 C). A máquina não é muito inteligente pois só aceita 2 tipos de moeda, a de 50 centavos (50 C) e a de 25 centavos (25 C), não aceitando a própria moeda de R$1,00 ou qualquer outra. A máquina só retorna ao usuário 2 itens possíveis, o refrigerante e, se aplicável, o único troco possível, de 25 centavos.

Assim, pode-se descrever a máquina de estados em questão com 2 bits de entrada. A combinação 00 significa que nenhuma moeda foi depositada naquele estado, 01 significa que a moeda de 25 C foi depositada e 10 significa que a moeda de 50 C foi depositada. Serão necessários 2 bits para representar as saídas do sistema. A saída 00 faz com que

a máquina não dê nada ao usuário. A saída 01 faz com que a máquina deposite em sua bandeja 1 lata do refrigerante. A saída 11 retorna ao usuário o refrigerante juntamente com o troco de 25 centavos.

Cada estado é nomeado com a quantidade, em centavos (C), de dinheiro que há naquele momento na máquina. Partindo do estado inicial (0 C), há 3 setas, uma para cada possível transição de estado. Como dito, cada uma dessas setas é acompanhada da entrada que permite aquela transição. A saída no estado 0 C é mostrada como 00, ou seja, o usuário ainda não recebe nada. Caso a entrada seja 00 (nenhuma moeda depositada), acompanha-se a seta 00, chegando-se ao próximo estado, que será o próprio estado 00. Caso a entrada seja 01 (25 C depositados), acompanha-se a seta 01, chegando-se ao novo estado da máquina, que será o 25 C. A última possibilidade é a entrada ser 10 (50 C depositados), acompanha-se a seta 10, que leva a máquina ao estado 50 C. Desta forma, a dinâmica de operação se desenvolve e é possível saber qual é o próximo estado da máquina observando-se o estado atual e as entradas naquele momento. Note que em quase todos os estados a saída permanece como 00, com exceção dos estados 100 C e 125 C. No estado 100 C a saída é 01 e é entregue ao usuário 1 lata de refrigerante e independente da entrada do usuário, o próximo estado será o estado inicial 0 C, isto é representado pela seta XX. No estado 125 C a saída é 11 e o usuário, como esperado, receberá, além de 1 lata de refrigerante, o troco de 25 C. Também aqui, independente da entrada, o próximo estado será o inicial.

O diagrama de estados, pela sua própria natureza gráfica, é a forma de representação mais intuitiva, o que a torna apropriada para etapas de interface com o usuário. Entretanto, para sua utilização na implementação de sistemas, é necessário que se traduza a máquina gerada graficamente para alguma outra representação apropriada, como a textual ou a tabular, para que essas informações possam ser processadas.

2.2 Representação Textual

A representação textual da máquina de estados utiliza a estratégia de explicitamente descrever todas as relações das combinações de estado atual e entrada com as combinações de saída e próximo estado. A máquina de estados pode então ser descrita com um pseudocódigo ou diretamente utilizando a sintaxe de alguma linguagem de programação específica, como C. Na Figura 4, pode-se ver a representação textual, em linguagem C, da mesma máquina de estados finitos do exemplo anterior, que está representada na Figura 3 sob a forma de diagrama de estados. Nesta representação atribuiu-se um número para cada estado. Os estados 0 C, 25 C, 50 C, 75 C, 100 C e 125 C têm valores 0, 1, 2, 3, 4 e 5, respectivamente. No código, inicia-se com estadoAtual = 0, com as duas saídas em 0, como esperado. A seguir, usa-se o valor do estadoAtual e dos valores de entrada para

determinar-se o valor de proximoEstado e os das saídas pertinentes. Por fim atualiza-se o valor de estadoAtual com o valor de proximoEstado definido.

Figura 4: Representação textual em linguagem C da máquina de estados do problema proposto.

```c
int entradas[2] = {0,0};
int saidas[2] = {0,0};
int estadoAtual = 0;
int proximoEstado;

while (1)
{
    switch(estadoAtual)
    {
        case 0:
            if((entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 1;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(entradas[1]))
            {
                    proximoEstado = 2;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 0;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            break;

        case 1:
            if((entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 2;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(entradas[1]))
            {
                    proximoEstado = 3;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 1;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            break;

        case 2:
            if((entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 3;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(entradas[1]))
            {
                    proximoEstado = 4;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 2;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            break;

        case 3:
            if((entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 4;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(entradas[1]))
            {
                    proximoEstado = 5;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            else if((!entradas[0])&&(!entradas[1]))
            {
                    proximoEstado = 3;
                    saidas[0] = 0;
                    saidas[1] = 0;
            }
            break;

        case 4:
                    proximoEstado = 0;
                    saidas[0] = 1;
                    saidas[1] = 0;
            break;

        case 5:
                    proximoEstado = 0;
                    saidas[0] = 1;
                    saidas[1] = 1;
            break;
    }
    estadoAtual = proximoEstado;
}
```

Fonte: Produzido pelo autor.

Essa representação, apesar de fácil implementação para pequenos casos, pode ser complicada para casos grandes, tendo portanto baixa escalabilidade. Note que para cada combinação de entradsa é necessário que se realize uma verificação, sendo que em muitos casos podem ocorrer verificações desnecessárias, o que se deseja evitar. Entretanto este tipo de representação pode ser muito útil se utilizado com linguagens de descrição de hardware, como VHDL, por exemplo.

2.3 Representação Tabular

A representação tabular consiste na determinação de uma tabela na qual se possa representar o estado atual do sistema e a entrada naquele momento. Para cada uma das combinações de estado atual e entrada define-se um conjunto de saídas e o próximo estado da máquina, de forma a descrever corretamente a dinâmica do sistema.

Utilizando o mesmo exemplo anterior, representado sob a forma do diagrama de estados da Figura 3, chega-se à Tabela 1, que é a forma tabular desta máquina de estados.

Tabela 1: Tabela descritiva da máquina de estados do exemplo proposto.

Estado Atual	**Entrada**	**Próximo Estado**	**Saída**
000	01	001	00
000	10	010	00
000	00	000	00
001	01	010	00
001	10	011	00
001	00	001	00
010	01	011	00
010	10	100	01
010	00	010	00
011	01	100	01
011	10	101	11
011	00	011	00
100	xx	000	00
101	xx	000	00

Fonte: Produzido pelo autor.

Nesta tabela cada nome de estado dá lugar a um conjunto de bits. Como existem 6 estados, são necessários 3 bits para representá-los. Os estados 0 C, 25 C, 50 C, 75 C, 100 C e 125 C têm valores binários de 000, 001, 010, 011, 100 e 101, respectivamente.

Na prática, os bits do estado atual e os das entradas físicas do sistema são concatenadas, dando origem a um conjunto de entradas maiores. Cada uma destas novas entradas representa uma combinação possível de entradas físicas e estado corrente. Da mesma forma, os bits que identificam o próximo estado da máquina são concatenados com os bits representativos das saídas do sistema. Este procedimento dá origem a Tabela 2, que representa exatamente a mesma máquina da Tabela 1. Esse será o tipo de representação utilizado para a confecção de um arquivo descritor do sistema, como será visto no Capítulo 4.

Tabela 2: Tabela compacta descritiva da máquina de estados do exemplo proposto.

Entradas	**Saídas**
00001	00100
00010	01000
00000	00000
00101	01000
00110	01000
00100	00100
01001	01100
01010	10001
01000	01000
01101	10001
01110	10111
01100	01100
100xx	00000
101xx	00000

Fonte: Produzido pelo autor.

Antecipa-se aqui a informação de que este tipo de representação é muito útil ao processamento destas informações, já que elas podem ser facilmente convertidas em expressões lógicas relacionando entradas e saídas. Isto é, na tabela, para uma determinada saída, o conjunto de entradas de cada linha em que esta saída esteja em estado alto, por exemplo, forma um mintermo da expressão lógica da saída considerada. A soma de todos os mintermos, ou seja, a soma dos mintermos de cada uma destas linhas, formará a expressão lógica completa daquela saída.

3 Uso de BDD para Representação de Máquinas de Estados Finitos

3.1 Os Diagramas de Decisão Binária (BDD)

O presente capítulo tem a pretensão de definir os diagramas de decisão binária, a fim de que o leitor entenda sua aplicação ao projeto. Sua implementação e teorias matemáticas não são abordadas pois fogem ao escopo deste trabalho. Para tais abordagens pode-se consultar as diversas referências a serem aqui citadas. Entre elas destaca-se o cultuado *The Art of Computer Programming, Vol. 4A, Combinatorial Algorithms, Part 1* de autoria de Knuth (2011).

> Diagramas de decisão binária (BDD) são magníficos, e quanto mais eu brinco com eles mais eu os amo. Por quinze meses eu estive como uma criança com um brinquedo novo, sendo agora capaz de resolver problemas que eu nunca imaginei que seriam tratáveis (KNUTH, 2008).

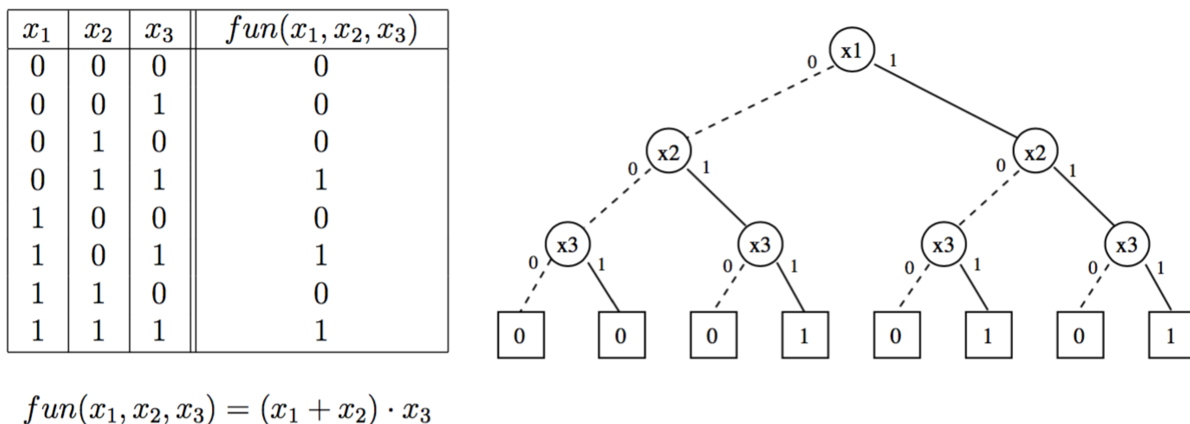
No Capítulo 2, viu-se que é possível representar uma máquina de estados finitos por uma tabela que relaciona logicamente as entradas e saídas do sistema. Tais tabelas constituem, então, expressões booleanas que descrevem tanto as saídas físicas do sistema, quanto as transições de estado. Entretanto, deve-se atentar ao fato de que a simples transformação desta tabela em expressões lógicas e a posterior verificação das saídas lógicas devem ser feitas com critério. Isso é necessário porque, por exemplo, o usuário pode ter descrito o sistema de maneira a gerar expressões lógicas redundantes e/ou não mínimas, o que aumentaria sem necessidade o tempo de cálculo das saídas do sistema. Há ainda o fato de que a ordem de verificação destas expressões é muito importante. Existe uma ordem ótima na qual é possível calcular a saída de uma expressão com o menor número de verificações possíveis. Neste contexto, faz-se necessária a utilização de uma ferramenta de manipulação das expressões booleanas descritoras do sistema, capaz de transformá-las a uma forma mínima e ordenada. Isso é, deve-se representar cada função booleana em sua forma minimizada, sem termos lógicos desnecessários. Deve haver uma ordenação na sequência de verificações de cada uma das entradas desta função para que a avaliação da espressão seja eficiente, ou seja, exija menos verificações. Uma das ferramentas mais indicadas pela literatura é o diagrama de decisões binárias (BDD, do inglês, *Binary Decision Diagrams*). De acordo com Knuth (2008, tradução nossa), BDD são "uma importante família de estrutura de dados que rapidamente se tornaram o método a ser escolhido para representação e manipulação de funções booleanas dentro de um computador".

Segundo Flores (1996, p.3), existem muitas formas de se representar funções booleanas. Tradicionalmente as formas utilizadas são tabelas verdade, mapas de Karnaugh ou

somas de produtos canônicas, porém, estas são computacionalmente impraticáveis, pois a representação de funções de n variáveis requer uma representação de tamanho 2^n. Além disto, estas têm a desvantagem de não terem uma forma canônica, ou seja, uma mesma função pode ter várias representações válidas diferentes, o que faz com que problemas de equivalência de funções ou de redundância sejam muito difíceis de se resolver. Assim se opta pela representação utilizando diagramas de decisão binária, expostos pela primeira vez por Lee (1959) na obra *Representation of Switching Circuits by Binary Decision Programs* e por Akers (1978) em *Binary Decision Diagrams.*

Um BDD representa uma função booleana $f(x_1, x_2, ..., x_n)$, como um grafo unidirecional, acíclico e com um nó de entrada, que é chamado de raíz. Neste tipo de diagrama, todo nó não terminal representa a avaliação de uma variável x_i da função f. De cada um destes nós sempre saem dois ramos, que representam a variável em questão ter valor de entrada atribuído 0 ou 1 (neste trabalho, estes valores são às vezes chamados de estado alto, verdadeiro e *high* ou estado baixo, falso e *low*). Os nós terminais indicam um dos valores possíveis para a função, 0 ou 1. A Figura 5 exibe a representação de uma função BDD de três variáveis, $fun(x_1, x_2, x_3) = (x_1 + x_2).x_3$. Nesta figura o ramo que leva um nó a seu nó filho de estado baixo é pontilhado, enquanto o ramo que o leva a seu nó filho de estado alto é contínuo.

Figura 5: BDD para uma função f de 3 variáveis.



x_1	x_2	x_3	$fun(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$fun(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$

Fonte: (FLORES, 1996, p. 3)

Conhecendo todos valores de entrada atribuídos às variáveis de uma função do tipo, pode-se facilmente determinar o valor resultante da função devido a essas atribuições percorrendo-se o grafo BDD. O percurso inicia sempre no nó raiz e finaliza em um dos dois tipos nós terminais, o nó 0 ou o nó 1. A cada nó não terminal, decide-se qual dos

dois caminhos tomar, pelo valor de entrada da variável sendo avaliada naquele nó. Ao final, chegar-se-á a um nó terminal, que será o resultado da função para aquele conjunto de entradas.

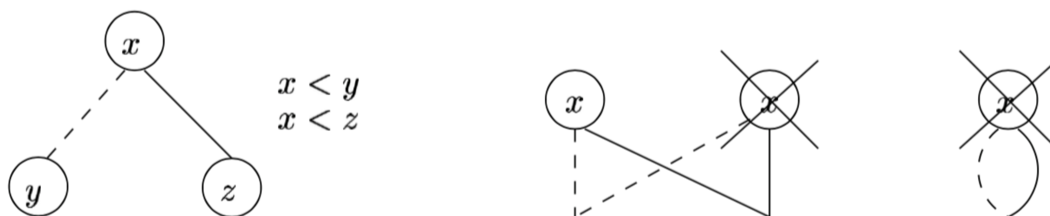
O diagrama BDD da Figura 5 não está em sua forma canônica. Bryant (1986), citado por Flores (1996, p. 4), introduziu restrições na ordenação das variáveis e também algoritmos para transformar um BDD em um diagrama de decisão binária ordenado e reduzido (ROBDD, do inglês, *Reduced Ordered Binary Decision Diagram*). Como os ROBDD são a forma mais habitual de utilização dos BDD, ao mencionar-se BDD, a partir daqui, faz-se referência a um ROBDD.

Andersen (1997, p. 12), define tais restrições, dizendo que um BDD será ordenado e reduzido se através de todos os caminhos do grafo as variáveis respeitarem uma dada ordem $x_1 < x_2 < x_3 < ... < x_n$. Isto é, seguindo qualquer caminho pelo grafo, cada variável só é avaliada uma única vez. Diz ainda que um BDD ordenado é reduzido se respeitar as seguintes propriedades:

- **Singularidade:** Não há dois nós distintos que representem a mesma variável, e tenham filhos de estado baixo e de estado alto iguais. Caso isso aconteça eles são exatamente o mesmo nó.
- **Não redundância:** Não há um nó x que tenha um mesmo nó como seu filho de estado baixo e também de estado alto. Isto é, $low(x) \neq high(x)$.

A Figura 6 ilustra os conceitos de ordenação (à esquerda), singularidade (ao meio) e não redundância (à direita).

Figura 6: Propriedades necessárias a um ROBDD.

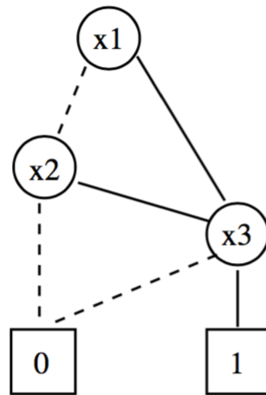


Fonte: (ANDERSEN, 1997, p. 12)

Tais propriedades, fazem com que o BDD seja uma representação compacta de expressões booleanas e que haja algoritmos eficientes para aplicar todos os tipos de operações lógicas em BDD. Isto se deve ao fato de que para qualquer função $f : B^n \rightarrow B$, existe exatamente apenas um BDD que a represente. Assim, é possível, em um tempo constante, avaliar se uma função é verdadeira ou falsa.

Após a aplicação de algoritmos de ordenação e restrição que, como dito, não serão abordados, o BDD da Figura 5 chega a sua forma mínima e canônica, exibida na Figura 7.

Figura 7: BDD ordenado e reduzido para uma função f de 3 variáveis.



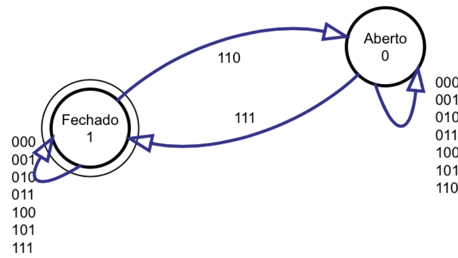
Fonte: (FLORES, 1996, p. 5)

3.2 Representando Máquina de Estados Finitos

Tendo sido definida a representação tabular de uma máquina de estados no capítulo 2, sua representação em BDD é intuitiva. Basta percebermos que cada diagrama BDD, como dito, representa uma função booleana. Como cada bit de saída da máquina de estados constitui uma função booleana, pode-se representar cada uma destas saídas por um BDD.

Observe o diagrama de estados da Figura 8. Esse diagrama representa a máquina de estados de uma catraca simples. A catraca tem 2 estados (s), seu estado inicial é Fechado (s = 0), com saída da trava em 1. A catraca tem uma senha de entrada de 3 bits (e2,e1,e0). Quando a catraca está fechada e digita-se a senha 110, ocorre uma transição para o estado Aberto (s = 1), levando a saída da trava (T) para 0, qualquer outra senha mantém a catraca fechada. Quando a catraca está aberta, a senha de entrada 111 a fecha, levando sua trava de volta a 1, qualquer outra senha mantém a catraca aberta. Tem-se então 3 bits de entradas físicas do sistema e necessita-se de mais 1 bit para representar os 2 estados do sistema. A saída da trava é representada por apenas 1 bit. Chega-se portanto a representação tabular desta máquina de estados, exibida na Tabela 3, na qual o próximo estado do sistema é definido para cada combinação realizável de estado atual e entradas físicas do sistema. Nota-se que na representação da máquina de Moore as saídas só dependem do estado atual do sistema. A saída (estado da trava T) exibida para cada combinação de entradas não é a saída do estado corrente e sim a saída associada ao

Figura 8: Diagrama de estados de uma catraca simples.



Fonte: Produzido pelo autor.

próximo estado do sistema, não sendo portanto a representação tabular exata da máquina de Moore e sim a representação tabular que é utilizada no trabalho para a montagem dos BDD.

Tabela 3: Representação tabular da máquina de estados de uma catraca simples.

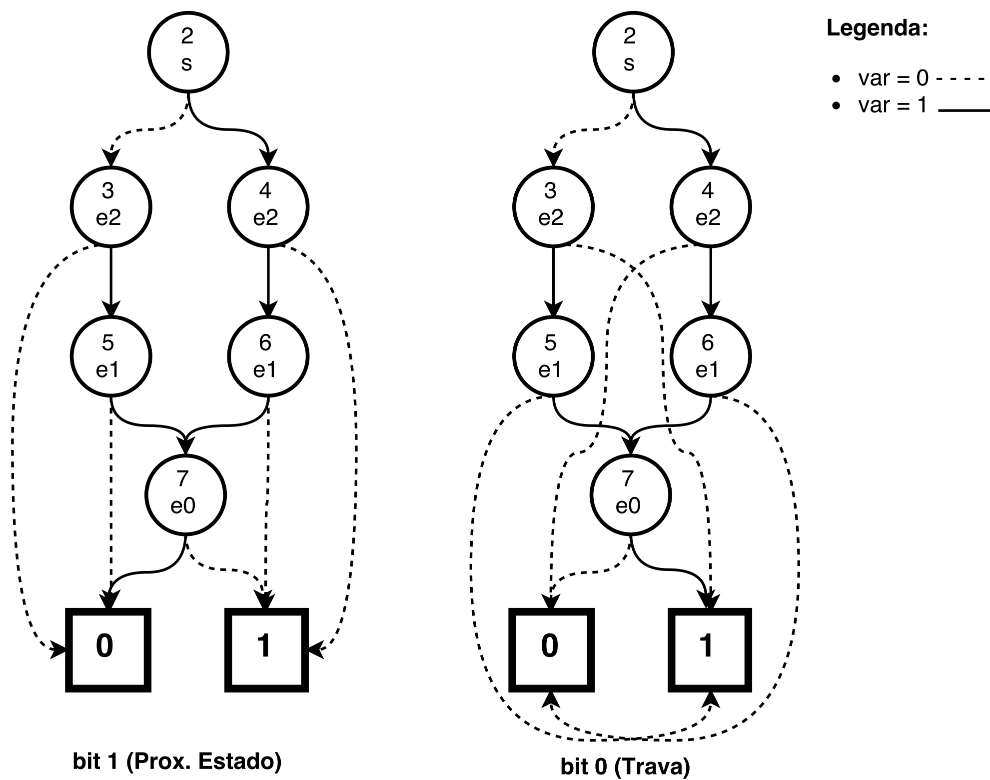
Entradas				Saídas	
s	e2	e1	e0	s	T
0	1	1	0	1	0
0	0	0	0	0	1
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	0
1	1	1	0	1	0

Fonte: Produzido pelo autor.

Gera-se então 1 BDD representando a função booleana de próximo estado da máquina (bit 1 da saída), e outro representando o estado da trava (bit 0 da saída). Estes dois BDD são exibidos na Figura 9. Aqui, o nó raíz tem número 2, e os nós terminais números 0 e 1, representando as saídas 0 e 1, respectivamente, das funções binárias por eles avaliada. Note que para se avaliar o próximo estado da máquina basta saber o estado corrente (s) e os 3 bits (e1,e2,e0) que representam a senha e percorrer o grafo até se

chegar ao nó terminal. Veja, por exemplo, que se a máquina está fechada (s = 0) e a senha 110 $(e2, e1, e0) = (1, 1, 0)$, tem-se um conjunto de entradas (0,1,1,0). Se percorre-se o grafo do próximo estado (bit 1) e toma-se os caminhos 0,1,1 e 0 em sequência, chega-se ao nó terminal 1, que é o próximo estado da máquina, ou seja, Aberto. O mesmo vale para a avaliação do estado da trava, ao percorrer-se seu grafo (Saída, bit 0) utilizando os caminhos (0,1,1,0), chega-se ao nó terminal 0, ou seja, trava desativada. Se conhece-se o estado corrente e as entradas da máquina de estados, pode-se utilizar o BDD para a determinação do próximo estado e da saída da máquina. É possível, então, representar qualquer máquina de estados finitos fazendo uso de BDD.

Figura 9: BDD de uma máquina de estados finitos.



Fonte: Produzido pelo autor.

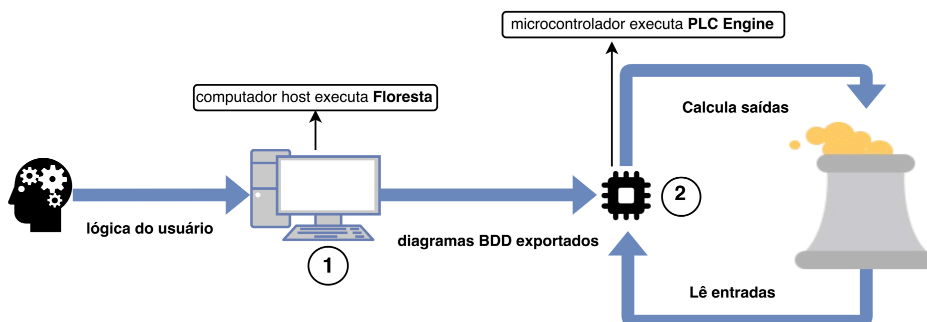
4 Implementação

Este capítulo trata da implementação de software desenvolvido no projeto. Todo o software foi desenvolvido em linguagem de programação C (C99). Esta linguagem é compilável para a maioria dos microcontroladores atualmente no mercado e conta com estruturas de dados suficientes para a aplicação. Além disso, a comunidade de desenvolvimento da linguagem já é grande e consolidada, o que permite encontrar-se diversas bibliotecas úteis ao desenvolvimento deste projeto. Tais bibliotecas serão aqui referenciadas e suas funções pertinentes ao projeto serão tratadas do ponto de vista funcional. Os programas foram escritos e testados na IDE Xcode da Apple.

A estratégia de programação adotada foi a de refinamento sucessivo, ou seja, a criação de várias funções simples para cada sub-tarefa de uma tarefa maior. Isto facilita o processo de criação dos programas e depuração de erros. Assim, cada função aqui, implementa uma ou poucas ações simples, juntas tais funções implementam ações mais complexas.

O software desenvolvido é dividido em 2 programas principais. O primeiro deles é o software Floresta, que deve ser executado em uma máquina host de desenvolvimento, onde o usuário implementará a máquina de estados finitos do sistema a ser controlado. Este é então responsável pela absorção da lógica do usuário e a criação de estruturas de dados, representando as estruturas BDD, que são exportadas. O segundo programa é o que de fato será embarcado em algum microcontrolador. Ele é responsável por importar as estruturas criadas e utilizá-las para descrever a lógica pretendida pelo usuário. Este último será muitas vezes neste trabalho referenciado como PLC Engine. Uma visão genérica da divisão dos dois programas pode ser vista na Figura 10.

Figura 10: Diagrama geral dos programas desenvolvidos.

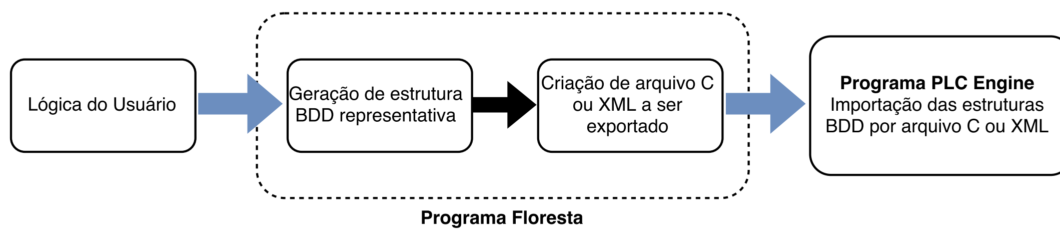


Fonte: Produzido pelo autor.

4.1 Programa Host, Floresta

O programa que será executado em um computador host, nomeado **Floresta**, foi desenvolvido para ser executado no ambiente de desenvolvimento do usuário. Isto é, após a síntese da lógica do sistema o usuário irá utilizar Floresta para criar as estruturas que alimentarão o sistema embarcado responsável por controlar o processo em si. Sendo mais claro, o programa tem como objetivo implementar as expressões lógicas do usuário em diagramas de decisão binária ordenados e reduzidos ou simplesmente diagramas de decisão binária (BDD), através da criação de árvores BDD que as representem. Em seguida deve transformá-las em estruturas de dados que possam ser facilmente exportadas para que o programa Engine, eventualmente embarcado, possa importá-las. A Figura 11 nos mostra o diagrama genérico do software Floresta.

Figura 11: Diagrama genérico do software Floresta.



Fonte: Produzido pelo autor.

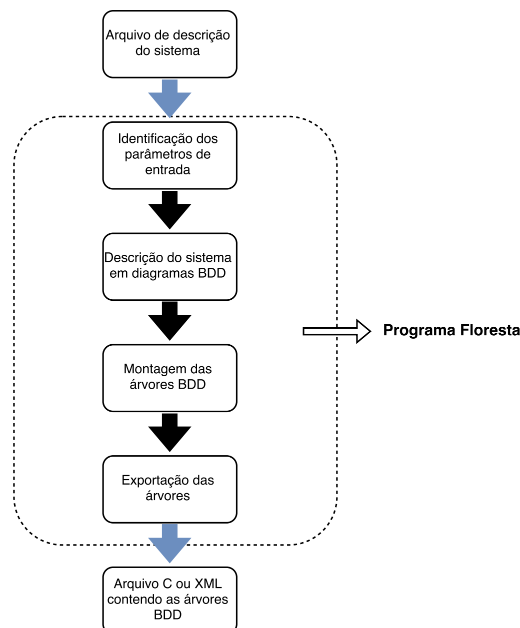
Foram adotadas duas abordagens de exportação. Na primeira, Floresta, deve ser capaz de gerar um arquivo .C, aqui nomeado **estufa.c**, que guardará as árvores do sistema. Este programa descreverá *structs* devidamente preenchidas. Este arquivo .C poderá então ser incluído através da diretiva **#include "estufa.c"** no programa Engine, possibilitando acesso direto às arvores. Esta abordagem é a mais simples e a que gasta menos recurso (memória) mas firma o compromisso de descrever a lógica no momento de compilação de Engine. Isto é, quando Engine for compilado para o microcontrolador, o arquivo **estufa.c** será incluído e a lógica ali descrita, em primeira instância, não poderá ser alterada. Esta abordagem é ideal para sistemas mais simples nos quais a lógica que deve ser descrita é fixa e não precisará de alterações constantes.

Na segunda abordagem, o programa deve gerar um arquivo .XML, aqui nomeado **florestarecuperada.xml**. O arquivo XML conterá uma "floresta", composta por diversas árvores que formam o sistema. Tal arquivo .XML será lido por Engine e será usado para preencher estruturas locais, similares às estruturas criadas estaticamente em **estufa.c**. O arquivo XML tem um *overhead* maior que o .C, ou seja, ele utilizará um maior espaço de memória para armazenar a mesma quantidade de informação útil. Além disso, o sistema

deve contar com recursos suficientes para armazenar este tipo de arquivo. Um outro ponto é que aqui o acesso não pode ser feito diretamente, ou seja, Engine terá que executar funções para, como dito, absorver o arquivo .XML e preencher as estruturas a serem acessadas. A vantagem é a possibilidade de atualização da lógica descrita com a simples substituição do arquivo XML e um novo preenchimento das estruturas, o que pode ser feito de maneira dinâmica, em tempo de execução. Esta abordagem é ideal para lógicas suscetíveis a mudanças e para sistemas que contam com interface de comunicação para transferência e atualização do arquivo XML e para sistemas com interface com dispositivos de armazenamento de arquivos, como cartões SD.

O funcionamento de Floresta compreende algumas tarefas básicas, que podem ser vistas na Figura 12. Primeiramente deve-se ler um **arquivo de descrição do sistema**, para se ter acesso à lógica do usuário. Em seguida deve-se fazer a **identificação dos parâmetros de entrada do sistema**, para que se extraia do arquivo descritor informações básicas do sistema para o próximo passo. Em sequência há a **descrição do sistema em diagramas BDD**, onde os BDD serão gerados a partir do arquivo descritor. Feito isso há a **montagem das árvores BDD**, onde serão construídas novas estruturas, chamadas árvores BDD, a partir dos diagramas construídos para possibilitar a exportação das estruturas BDD e facilitar a futura leitura por PLC Engine. Este conjunto de árvores é, muitas vezes no texto, referido como "floresta". Por fim há a **exportação das árvores BDD**, onde será gerado um arquivo C ou XML contendo tais árvores BDD representativas do sistema. Cada uma destas etapas é descrita nas subseções a seguir.

Figura 12: Diagrama detalhado do software Floresta.

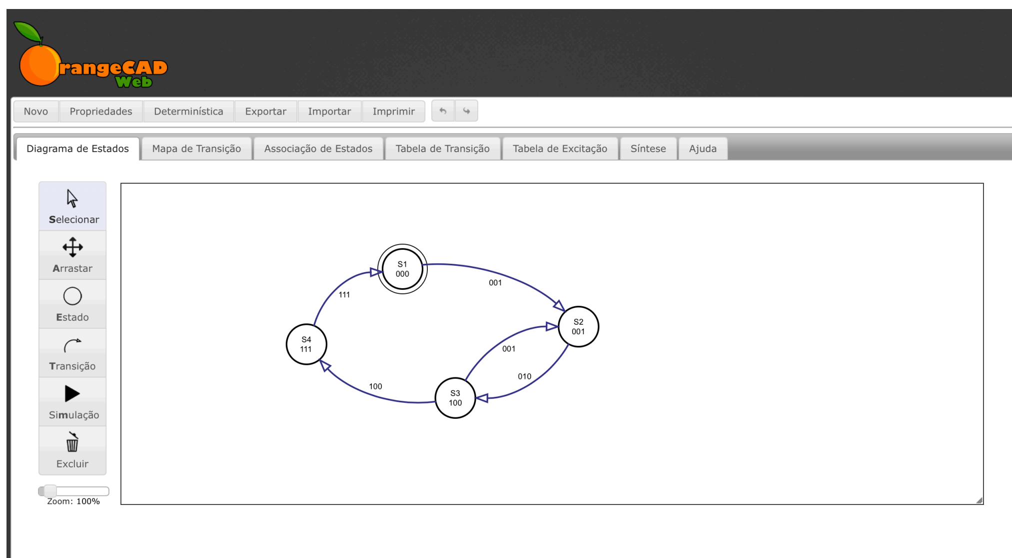


Fonte: Produzido pelo autor.

4.1.1 Arquivo de descrição do sistema

Para iniciar a implementação aqui desejada, necessita-se de algum recurso que seja capaz de descrever o processo que o usuário pretende controlar. Para isso o presente trabalho utiliza o software **OrangeCad Web** desenvolvido no projeto de graduação de Ribeiro (2011), pela Universidade Federal do Espírito Santo. Tal software dispobiliza ao usuário uma interface gráfica para construção da máquina de estados finitas desejada, como pode ser visto na Figura 13.

Figura 13: Tela do Software OrangeCad.



Fonte: Produzido pelo autor.

Esta ferramenta permite que sejam gerados arquivos de saídas em dois formatos diferentes, VHDL e ESPRESSO. Assim, neste trabalho, utilizou-se o arquivo no formato ESPRESSO gerado como fonte da descrição do sistema. O arquivo ESPRESSO representa a máquina de estados implementada graficamente pelo usuário (em diagrama de estado) na forma tabular. Tendo ambas as formas de representação sido discutidas no Capítulo 2.

4.1.1.1 Arquivo ESPRESSO

O formato ESPRESSO possibilita a descrição de máquinas de estados finitos. Um exemplo de arquivo deste tipo pode ser visto na Figura 14. Este arquivo representa a máquina de estados do exemplo proposto no Capítulo 2. Note que inicialmente o número de entradas e saídas são escritos após os indicadores **.i** e **.o**, respectivamente. Em seguida há a descrição das expressões lógicas propriamente dita. Cada linha traz uma relação entre entradas e saídas diferentes, sendo que o conjunto de colunas da esquerda indica o estado das entradas nesta relação (0 estado baixo, 1 estado alto) e o conjunto direita

indica o estado de cada uma das saídas do sistema (0 estado baixo, 1 estado alto e - estado qualquer). O indicador **.e** sinaliza o fim do arquivo ESPRESSO. Estas relações lógicas são exatamente a representação tabular compacta de uma máquina de estados, discutidas, como dito, no Capítulo 2. Destaca-se a recomendação de definição do estado de qualquer bit em 0 para os casos onde o valor não importar (x - *don't care*), no ato de confecção do arquivo descritor. Isso foi feito no arquivo da Figura 14 e é recomendado para compatibilidade com Floresta.

Figura 14: Exemplo de um arquivo descritor no formato ESPRESSO.

```
# ================================================
#   UFES - Universidade Federal do Espirito Santo
#   Descricao do circuito no formato ESPRESSO
#   Arquivo gerado automaticamente por OrangeCAD Web
#   November 13, 2015 at 22:36:54 GMT-2
# ================================================
.i 5
.o 5
00001 00100
00010 01000
00000 00000
00101 01000
00110 01100
00100 00100
01001 01100
01010 10001
01000 01000
01101 10001
01110 10111
01100 01100
10000 00000
10100 00000
.e
```

Fonte: Produzido pelo autor.

4.1.2 Identificação dos parâmetros de entrada do sistema

Em posse de um arquivo descritor da máquina de estados, o software Floresta deve ser capaz de lê-lo, extraindo as informações necessárias. É preciso que sejam identificados os parâmetros de entrada do sistema, isto é, a quantidade de entradas (nEntradas), a quantidade de saídas (nSaidas) e a quantidade de expressões lógicas necessárias à descrição do processo (nExpressoes). Tais informações são extraídas pela função denominada **setaParametros(char *caminhoEspresso)**.

A função, como mostrado acima, recebe como parâmetro uma String que indica o caminho para o arquivo descritor do tipo Espresso. A rotina lê o arquivo indicado e inicializa corretamente as variáveis nEntradas, nSaidas e nExpressoes. No exemplo da Figura 14 esses valores são 5, 5 e 14, respectivamente.

4.1.3 Descrição do sistema em diagramas BDD

Com os parâmetros do sistema definidos, pode-se iniciar a construção do BDD correspondente aquele sistema. Para criar e manipular tais diagramas foi utilizada a biblioteca **BuDDy**, desenvolvida por Lind-Nielsen et al. (2001). A escolha baseou-se na compatibilidade com a linguagem adotada no projeto (C), disponibilidade de documentação e de repositório online da biblioteca com correções e aplicações diversas, disponível no repositório de Lind-Nielsen (2007).

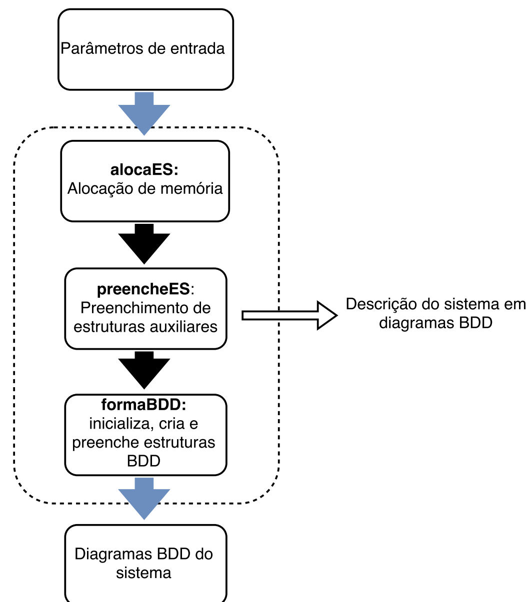
Através da inclusão da biblioteca BuDDy (**bdd.h**), temos acesso a todas as suas funções e estruturas de dados. Aqui, cada nó de uma árvore é do tipo bdd. O primeiro passo é a declaração de dois ponteiros, **bdd *entradasBDD, *saidasBDD**, que apontarão para vetores do tipo BDD, tipo definido pela biblioteca mencionada, representando as entradas e saídas BDD. Estes vetores serão alocados dinamicamente de acordo com os parâmetros do sistema já estabelecidos.

Feita as declarações necessárias e com os parâmetros de entrada do sistema (número de entradas, número de saídas e número de expressões), a construção dos diagramas de decisão binária pode ser realizada em algumas etapas, exibidas na Figura 15. Primeiramente deve-se alocar memória para as estruturas auxiliares e para os próprios diagramas BDD, isto é feito pela função **alocaES**. Em sequência, a partir do arquivo descritor do sistema, a função **preencheES** realiza o preenchimento das estruturas auxiliares (AUXentradas e AUXsaidas) que conterão os estados de entrada e saída de cada expressão lógica descrita pelo arquivo. Por fim, a função **formaBDD** cria, inicializa e preenche os diagramas BDD já alocados, baseando-se nos dados guardados nas estruturas auxiliares. Mais detalhadamente, primeiro deseja-se armazenar o conjunto de colunas de entradas e saídas do arquivo ESPRESSO que descrevem as relações lógicas do sistema. Para isso, a função **alocaES** aloca dinamicamente memória para armazenar tais informações e inicializa os ponteiros de ponteiros **int** AUXentradas, AUXsaidas** com os endereços dos segmentos de memória alocados para a construção das matrizes desejadas. A função também aloca memória para entradasBDD e saidasBDD.

O próximo passo é feito na chamada da função **preencheES**. A rotina lê o arquivo ESPRESSO, preenchendo as matrizes AUXentradas e AUXsaidas com os valores de entradas e saídas das expressões lógicas, ou seja, copiando os conjuntos de entradas e saídas, que descrevem as relações lógicas, para matrizes internas do programa (AUXentradas e AUXsaidas), o que facilita a futura utilização destas informações.

Com todos os segmentos de memória alocados e todas as informações do arquivo ESPRESSO armazenadas, o programa executa **formaBDD**. Inicialmente a rotina chama a função **bdd_init(nodenun, cachesize)** para inicializar o pacote de tratamento BDD, como traz Lind-Nielsen (2007). O parâmetro nodenun configura o número inicial de nós

Figura 15: Descrição do sistema em diagramas BDD.



Fonte: Produzido pelo autor.

BDD que serão utilizados e cachesize configura o tamanho dos caches usados por cada operador BDD. Estes caches serão utilizados por funções como **bdd_apply**, utilizada para aplicar operações lógicas entre nós. A Tabela 4 traz as sugestões da documentação para os valores destes parâmetros. Um ponto importante é que uma opção conservadora por uma quantidade pequena de nós pode reduzir a performance de utilização da biblioteca pois aumentará o número de *garbage collections* necessárias, ou seja, aumentará a frequência de execução do procedimento de limpeza de nós da memória não utilizados para realocação de novos nós. Se preciso, o pacote automaticamente irá aumentar a quantidade de nós utilizados.

Tabela 4: Recomendação de parametrização da função bdd_init.

Aplicação	**nodenun**	**cachesize**
Pequenos exemplos teste	1000	100
Pequenos exemplos	10000	1000
Exemplos médios	100000	10000
Exemplos grandes	1000000	variável

Fonte: Lind-Nielsen (2007, tradução nossa)

.

Após a inicialização efetuada, formaBDD chama a função **bdd_setvarnum(nEntradas)**, para que se defina o número **nEntradas** de variáveis utilizadas na manipulação dos bdd.

A criação dos nós de cada árvore é então feita através da função **bddith_ithvar(i)**,

que recebe como parâmetro o número **i** da variável avaliada naquele nó e retorna a referência para o nó criado na tabela de nós BDD de implementação interna da biblioteca. Esta função deve então ser usada para atribuir referências a cada um dos nós do tipo bdd declarados inicialmente para cada uma das variáveis existentes, ou seja, para inicializar cada posição do vetor **entradasBDD**.

Pode-se agora efetivamente construir os diagramas. Para isso são utilizadas três funções básicas da biblioteca bdd: **bdd bdd_not(BDD)**, **bdd bdd_and(BDD, BDD)** e **bdd bdd_or(BDD, BDD)**. A função bdd_not nega um nó BDD declarado, a função bdd_and aplica um and lógico entre dois nós BDD e a função bdd_or aplica um or lógico entre dois nós BDD.

Das três funções básicas nativas derivou-se algumas outras funções. Primeiramente, a função **bdd * bddAnd(int **AUXentradas, int i)**, recebe a matriz de entradas AUXentradas e o valor inteiro i, que especifica uma linha desta matriz. A função faz então um and lógico entre todos os nós de entradasBDD, negando o nó referente a uma determinada entrada caso seu valor em AUXentradas seja 0. Isto é feito principalmente porque a função bdd_and, nativa da biblioteca bdd, só aceita 2 nós como entrada. Para exemplificar, observe o arquivo ESPRESSO exibido na Figura 16. Este traz a descrição de um sistema simples que implementa uma porta "ou exclusivo - XOR"de 3 entradas.

Figura 16: Arquivo descritor ESPRESSO para XOR de 3 entradas.

```
.i 3
.o 1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 0
.e
```

Fonte: Produzido pelo autor.

Caso apliquemos bddAnd à terceira linha da matriz AUXentradas preenchida através deste arquivo, a função bddAnd executaria a seguinte sequência:

1. bdd_and ao valor negado do primeiro nó de entradasBDD com o segundo nó de entradasBDD

2. bdd_and entre o BDD resultante da operação anterior e o valor negado do terceiro nó de entradasBDD

Em seguida é feita uma segunda derivação das funções nativas em **bdd * bddOr(bdd **AUXsaidas, bdd **AUXentradas, int j)**. Esta função recebe como parâmetros as matrizes AUXsaidas e AUXentradas e o inteiro j, que especifica uma coluna da matriz AUXsaidas, ou seja, uma das nSaidas do sistema. A rotina faz uma varredura nos termos de todas as nExpressoes linhas da "j-ésima"coluna da matriz AUXsaidas. Quando encontra um encontra um valor 1 em uma eventual linha x, significa que aquela combinação de estados das entradas, que podem ser vistos em AUXentradas, leva a saída j para seu estado alto. Assim, a função bddAnd, descrita anteriormente, é aplicada para a linha x de AUXentradas. Isto é feito para todas as linhas que têm valor 1 em AUXsaidas. Em seguida aplica-se bdd_or entre todos os BDD resultantes destas operações de bddAnd. O resultado final da execução de bddOr é a referência para um diagrama BDD completo para uma determinada saída j. A função formaBDD aplica então bddOr a todas as nSaidas, inicializando todas as posições de saidasBDD.

Para exemplificar, o mesmo arquivo XOR de 3 entradas da Figura 16, tem algumas de suas linhas destacadas na Figura 17. A função bddOr identificaria os estados altos da coluna AUXsaidas, marcados em azul. Em seguida, faria um "e lógico" com as entradas de cada uma das linhas marcadas em vermelho. Por fim, faria um "ou lógico"com as expressões resultantes do "e lógico"anterior e atribuiria isto a um nó de saída em saidasBDD.

Figura 17: Arquivo descritor ESPRESSO para XOR de 3 entradas, com marcações.

```
.i 3
.o 1
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 0
.e
```

Fonte: Produzido pelo autor.

Ao final, saidasBDD tem cada uma das relações lógicas entre entradas e saídas mapeadas em diagramas BDD. Se, por exemplo, o sistema fosse simplesmente regido por uma expressão de "Ou Exclusivo (XOR)" de 3 entradas mostrada anteriormente, sua tabela representativa do diagrama BDD gerado seria a da Figura 18, onde a primeira coluna representa o número de referência na tabela de cada nó e a segunda coluna representa o número da variável (var), ou entrada, sendo avaliada no nó. A terceira e quarta colunas representam qual é o próximo nó se a entrada avaliada estiver, respectivamente, em seu estado 0 e 1. Percebe-se que os números dos nós de referência não são sequenciais, isto porque a biblioteca **BuDDy** mantém uma tabela de nós de referência única para toda

a execução, reaproveitando muitas vezes nós idêntico. Além disso, destaca-se que o valor em si do número de referência do nó não faz nenhuma diferença, contanto que sempre se guardem as mesmas referências para cada nó específico.

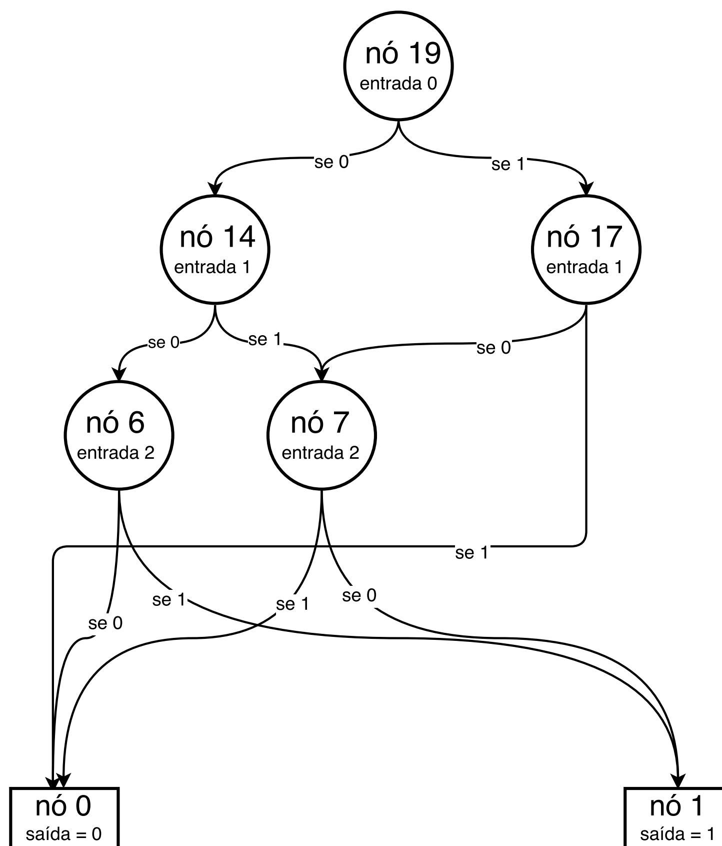
Figura 18: Diagrama BDD de um sistema XOR de 3 entradas em tabela.

```
ROOT: 19
[    6]   2:   0   1
[    7]   2:   1   0
[   14]   1:   6   7
[   17]   1:   7   0
[   19]   0:  14  17
```

Fonte: Produzido pelo autor.

A tabela BDD exibida constitui o diagrama, ou árvore, de decisão binária ordenado e reduzido exibido na Figura 19.

Figura 19: Diagrama BDD de um sistema XOR de 3 entradas em árvore.



Fonte: Produzido pelo autor.

A redução e a ordenação dos diagramas são feitas pelas próprias funções nativas. Vale ressaltar que, sempre ao aplicar uma destas operações lógicas, deve-se aplicar

bdd_addref, nativa da biblioteca bdd, ao BDD resultante das operações e então passar o BDD resultante de bdd_addref em seu lugar. Isto é, se o usuário quiser aplicar um and lógico entre dois nós, A e B, e fazer com que bdd_exemplo receba isto, ao invés de se fazer **bdd bdd_exemplo = bdd_and(A, B)**, deve-se fazer **bdd bdd_exemplo = bdd_addref(bdd_and(A, B))**. Isto porque, como nos traz a documentação, a biblioteca bdd precisa incrementar o *reference count* daquele nó antes de qualquer outra operação, evitando que ele seja descartado no processo de *garbage collection*.

Viu-se que após a criação de nós BDD representando variáveis com bddith_var pode-se, através das operações bdd_not, bdd_and e bdd_or, construir qualquer operação lógica. Pode-se ainda, através das operações extendidas bddAnd e bddOr, formar-se um sistema completo de expressões lógicas para cada saída do sistema, mapeando uma tabela aqui representada pelo arquivo ESPRESSO e posteriormente pelas matrizes AUXentradas e AUXsaidas. Tais saídas são representadas cada uma por um nó BDD raiz, sendo cada nó armazenado pelo vetor saidasBDD, como foi mostrado.

Cada nó do tipo BDD é representado por um número de referência na tabela de nós criada pela biblioteca para manipular os diagramas BDD. Os nós representam a avaliação de uma variável de entrada específica e apontam para outros dois outros nós BDD, um nó caso sua variável representativa tenha estado 0 e outro nó caso tenha estado 1. Isso acontece naturalmente, como em um diagrama BDD, desde o nó raiz até que se chegue nos nós terminais 0 ou 1. A biblioteca permite que se apliquem várias funções sobre estes nós para que essas informações sejam extraídas, sendo as principais para este escopo:

- **int bdd_var(bdd)**, que retorna a variável de um nó BDD
- **int bdd_low(bdd)**, retorna o nó filho de estado 0 de um nó BDD
- **int bdd_high(bdd)**, retorna o nó filho de estado 1 de um nó BDD

No exemplo da Figura 18 e da Figura 19, o nó raiz tem valor 19 (primeira coluna da tabela) e representa a variável 0 (segunda coluna da tabela). Seu nó filho caso a variável 0 assuma valor Low ou baixo (0) é o 14 e caso assuma valor high ou alto (1) é o 17. A tabela mostra todas as relações do diagrama até que se chegue nos nós terminais 0 ou 1.

4.1.4 Montagem das árvores BDD

Apesar de a biblioteca possibilitar o acesso a todos os dados da tabela através das funções mostradas anteriormente, é conveniente que essas informações sejam copiadas para uma estrutura interna, mais simples e mínima para assim facilitar o desenvolvimento futuro. Isto é importante para que, por exemplo, o segundo software, que será embarcado, seja totalmente independente da biblioteca bdd aqui utilizada, só sendo essa necessária

no software *host*, Floresta. Desta forma, criou-se um tipo próprio para representar um nó. Este foi denominado **treeNode**. O diagrama BDD será então reconstruído tendo cada nó representado por uma variável do tipo treeNode, a união destes treeNodes é aqui chamada de árvore, em alusão a referência de um nó a outro da raiz ao nó terminal, tal qual acontece em uma árvore.

4.1.4.1 O tipo treeNode

Este tipo deve conter o mínimo suficiente para representar um nó, para que desta forma seja possível armazenar todas as informações da tabela de implementação dos diagramas BDD mostrados anteriormente. Assim o tipo idealizado contém quatro inteiros, como pode ser visto na Figura 20.

Figura 20: Definição do tipo treeNode.

```
typedef struct treeNode
{
    int node;
    int var;
    int lowNode;
    int highNode;
} treeNode;
```

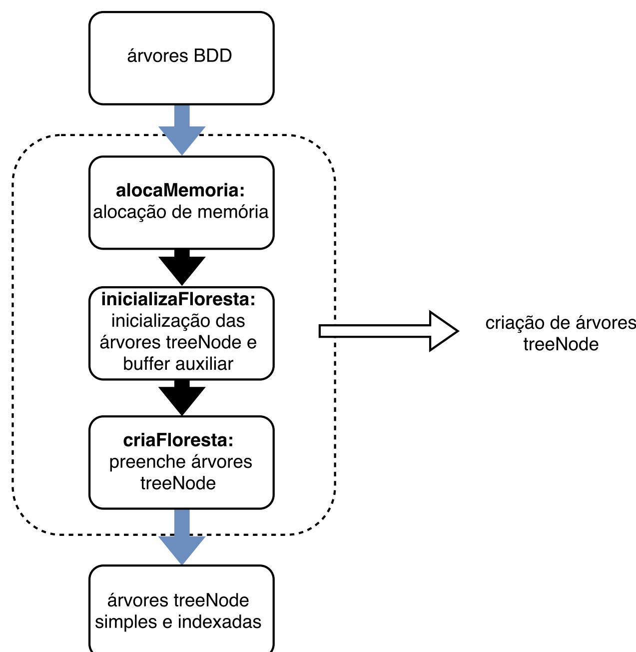
Fonte: Produzido pelo autor.

- **node**: representa o valor do nó.
- **var**: representa a variável que aquele nó está avaliando.
- **lowNode**: representa o nó que sucede o nó atual, caso a variável avaliada esteja em seu estado 0 (low ou baixo)
- **highNode**: representa o nó que sucede o nó atual, caso a variável avaliada esteja em seu estado 1 (high ou alto)

4.1.4.2 Construção de árvores simples

A ideia aqui é a de usar as funções bdd_var, bdd_low e bdd_high para, desde o nó bdd raiz, verificar as informações (node, var, lowNode e highNode) de cada nó bdd, copiando-as para estruturas análogas treeNode. Desta forma, ao final do processo, teremos uma cópia funcional desses diagramas na estrutura própria aqui idealizada. Isto será feito através de algumas etapas, exibidas na Figura 21. Primeiramente aloca-se memória para as árvores treeNode com **alocaMemoria**, em seguida **inicializaFloresta** inicializa as árvores treeNode alocadas e também o *buffer* que auxiliará o preenchimento das árvores. Por último **criaFloresta** preenche as árvores alocadas.

Figura 21: Diagrama de criação de árvores treeNode.



Fonte: Produzido pelo autor.

Mais detalhadamente, na estratégia adotada, inspeciona-se inicialmente o nó raiz de um diagrama BDD e guarda-se suas informações em uma variável treeNode. Adiciona-se os valores de nó de seus filhos a um *buffer*. A seguir repete-se a operação de cópia de informações do nó bdd para uma estrutura treeNode com os nós bdd que constam no buffer. É importante verificar se um nó já foi adicionado ao *buffer* antes de fazê-lo pois pode haver nós que apontam para um outro nó em comum, o que causaria múltiplas cópias de um mesmo nó bdd para a árvore. Esta verificação é implementada pela função **int verificaBuffer(int *a, int n, int tamanho)**, que recebe como parâmetros a referência para o buffer 'a' a ser verificado, o valor 'n' do nó que que deve ser verificado e o tamanho do *buffer*. A função retorna 0 caso 'n' já esteja no *buffer* e 1 caso ainda não esteja. O ponteiro **int **buffers** guarda um *buffer* para cada uma das nSaidas do sistema. O ponteiro **int *jbff** guarda um inteiro de controle de buffer para cada uma das nSaidas. Este inteiro informa a próxima posição livre para armazenamento de nós em cada *buffer*.

Seguindo o diagrama exibido na Figura 21, é necessário, primeiramente, que se aloque memória para os nós treeNode a serem criados e para as estruturas auxiliares buffers e jbff. Isto é implementado pela função **alocaMemoria**. A rotina aloca memória para **treeNode **arvores**, que armazenará treeNodes formando árvores e também para treeNode **arvoresIndexadas que armazenará treeNodes formando uma nova espécie de árvore que será apresentada posteriormente. A função alocaMemoria utiliza uma outra função nativa da biblioteca bdd, **extern int bdd_nodecount(BDD)**, que retorna a

quantidade de nós de um diagrama BDD dado seu nó raiz. Ela então aloca memória para o vetor **int *tArvores** e, através de bdd_nodecount, armazena a quantidade de nós de cada diagrama BDD de saidasBDD, ou seja, o tamanho de cada uma das árvores a serem construídas.

Em seguida a função **inicializaFloresta** chama **inicializaArvores**, que inicializa os três primeiros nós de cada uma das nSaidas árvores. O primeiro nó é sempre reservado para representar o nó terminal 0, o segundo nó sempre representa o nó terminal 1 e o terceiro nó é sempre reservado para o nó raíz de cada árvore. Chama também a função **inicializaBuffer**, que para cada elemento de saidasBDD (que contém os nós raízes dos diagramas BDD gerados), verifica se os filhos (lowNode e highNode) destes nós raízes já foram adicionados ao *buffer*. Caso não tenham sido a função os adiciona e incrementa jbff da saída correspondente.

Pode-se então finalmente chamar **criaFloresta**, que executará **bdd2Arvores**. A função, como já introduzido, executa a seguinte sequência:

1. Copia as informações (node, var, lowNode e highNode) do primeiro nó armazenado no buffer para uma estrutura treeNode no vetor arvores.
2. Verifica se o lowNode deste nó encontra-se no buffer. Caso não se encontre, adiciona-o e incrementa o contador jbff.
3. Verifica se o highNode deste nó encontra-se no buffer. Caso não se encontre, adiciona-o e incrementa o contador jbff.
4. Repete o procedimento por um N número de vezes (valor armazenado em tArvores) para que todos os nós de uma determinada saída em saidasBDD sejam armazenados, formando uma árvore completa.
5. Repete o procedimento para todas as saídas do sistema (cada uma contém um *buffer*) formando todas as nSaidas árvores.

4.1.4.3 Construção de árvores indexadas

Após os procedimentos de construção de árvores simples relatados, tem-se todas as árvores, com todos os seus nós, armazenadas na matriz de treeNodes denominada "arvores". Cada árvore contém uma cópia exata das informações de node, var, lowNode e highNode das tabelas BDD geradas na etapa de construção dos diagramas BDD. O problema é que as estruturas treeNode não contém ponteiros para seus nós filhos (lowNode e highNode), contendo apenas seus valores de nó (node). Assim, caso se queira percorrer um dos diagramas desde seu nó raiz até seu nó terminal, para uma determinada entrada, tería-se que varrer toda a árvore procurando pelo nó filho a cada salto de nó. Esta busca

exaustiva não é desejada. Desta forma, para solucionar este problema sem termos que criar ponteiros na estrutura treeNode, adotaremos a estratégia simples de substituir o valor de node de cada nó por seu índice no vetor arvores, ou seja, por sua própria posição na árvore. Com esta medida o acesso a cada nó é intuitivo e direto, como pode se ver no exemplo a seguir:

Suponha que em um conjunto de árvores, esteja-se fazendo referência ao quarto treeNode da segunda árvore, isto é: arvores[2][4]. Seu nó filho de estado baixo (lowNode) é arvores[2][4].lowNode. Se, por exemplo, arvores[2][4].lowNode = 8, então para se acessar este nó basta fazer referência a arvores[2][8], ou seja, ao oitavo nó da mesma árvore (a segunda).

Esta transformação de um conjunto de árvores simples para este novo tipo de árvore, chamado arvoresIndexadas, é feito pela função **arvores2arvoresIndexadas**. Vale lembrar que a alocação de memória para conjunto de árvores já foi previsto e realizado anteriormente, juntamente com a alocação de memória para as árvores simples, pela função alocaMemoria. Para essa construção só é preciso a criação de mais uma função, **int indexSeeker(int node, treeNode* arvore, int n)**. Esta função recebe como parâmetros o nó (node) que se está buscando, a árvore (arvore) em que ele se encontra e o tamanho 'n' desta árvore. Ela realiza uma busca pela árvore, retornando o índice do nó buscado. Caso não encontre o nó seu valor de retorno é -1.

Assim arvores2arvoresIndexadas inicializa o novo conjunto de árvores, arvoresIndexadas, sendo que os valores de nodes, lowNode e highNode de cada treeNode dão lugar a índices retornados por indexSeeker. Os valores de var são mantidos. Finalmente tem-se um conjunto de árvores com informações mínimas e suficientes para se percorrer todo o caminho de um BDD, de sua raíz a seus nós terminais. Passa-se a chamar este conjunto de árvores que representam os BDD do sistema de Floresta.

4.1.5 Exportação das árvores BDD

De posse da Floresta representativa do sistema pode-se finalmente exportá-la, de forma que o software PLC Engine, a ser embarcado, possa recuperar as informações de todos os nós que compõem o sistema.

4.1.5.1 Exportando Floresta em um arquivo C

Como introduzido no começo deste capítulo, a primeira abordagem é a criação de um arquivo C. Este arquivo terá as árvores indexadas anteriormente construídas declaradas explicitamente. Desta forma o futuro software a ser embarcado pode facilmente incluí-lo e acessar suas informações diretamente, como se elas tivessem sido geradas em sua execução. Chama-se este arquivo de **estufa.c**, ou simplesmente estufa.

O primeiro passo é a criação da função **void plantaArvoreC(int iArvore, int nArvore, treeNode arvore[], FILE *fp)**. Esta recebe como parâmetro um conjunto de árvores arvore[], um inteiro iArvore que especifica uma das árvores do conjunto, o tamanho nArvore da árvore especificada (quantidade de nós) e a referência para o arquivo que será usado para escrever as árvores. Tal função abre o arquivo passado em modo de escrita e declara a árvore especificada como sendo o elemento iArvore de uma variável chamada arvoreC[iArvore], do tipo treeNode. Na Figura 22 pode-se ver um exemplo de resultado da utilização da função. Neste caso usou-se a rotina para escrever uma árvore que contém 9 nós e é a segunda árvore (arvoreC2) de um conjunto de árvores, isto é:

plantaArvoreC(2,9, arvores, "caminhodoarquivo/estufa.c")

Figura 22: Exemplo de escrita feita pela função plantaArvoreC.

```
treeNode arvoreC2[9] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,0,5},
 {4,1,6,7},
 {5,2,0,1},
 {6,3,0,1},
 {7,2,6,8},
 {8,3,1,0}
};
```

Fonte: Produzido pelo autor.

Em seguida, pode-se efetuar a exportação completa através da função criada **void plantaFlorestaC(treeNode *floresta[], int *tArvores, int nsaidas)**. Esta rotina recebe como parâmetros a matriz de treeNode, floresta, contendo todas as árvores a serem exportadas, o vetor de inteiros tArvores, contendo o tamanho de cada uma das árvores e o inteiro nsaidas, especificando o número de saídas do sistema, ou seja, o número de árvores existentes. A rotina abre um arquivo chamado **estufa.c**, o qual será utilizado para escrever as estruturas a serem exportadas. A primeira linha escrita é a de inclusão do arquivo *header* onde o tipo treeNode foi definido, para que se possa utilizá-lo. Em sequência plantaFlorestaC utiliza a função plantaArvoreC para escrever a inicialização para cada uma das nSaidas árvores existentes, isto é arvoreC1, arvoreC2... arvoreCnSaidas. Por fim a função declara um vetor de árvores (matriz de treeNode) e a inicializa com as árvores escritas anteriormente, isto é treeNode *florestaC[nSaidas] = arvoreC1, arvoreC2..., arvoreCnSaidas. A Figura 23 nos mostra o resultado da escrita de um arquivo do tipo. Neste caso o sistema continha 4 árvores, com 6,8,9 e 4 nós cada uma.

Figura 23: Exemplo de escrita feita pela função plantaFlorestaC.

```
#include "floresta.h"

treeNode arvoreC0[6] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,0,3},
 {3,1,0,4},
 {4,2,0,5},
 {5,3,0,1}
};
treeNode arvoreC1[8] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,0,3},
 {3,1,4,5},
 {4,2,0,6},
 {5,2,0,7},
 {6,3,0,1},
 {7,3,1,0}
};
treeNode arvoreC2[9] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,0,5},
 {4,1,6,7},
 {5,2,0,1},
 {6,3,0,1},
 {7,2,6,8},
 {8,3,1,0}
};
treeNode arvoreC3[4] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,1,0,3},
 {3,3,0,1}
};
treeNode *florestaC[4] = {
arvoreC0,
arvoreC1,
arvoreC2,
arvoreC3};
```

Fonte: Produzido pelo autor.

4.1.5.2 Exportando Floresta em um arquivo XML

Como já mencionado, a segunda abordagem de exportação é a criação de um arquivo XML. Para isto foi utilizada a biblioteca **Mini-XML** compilável para C padrão, desenvolvida e mantida por Sweet (2003). Partindo das funções e tipos trazidos pela biblioteca, criou-se uma nova biblioteca própria denominada **arvoreXML**, capaz de criar um arquivo XML a partir de uma matriz de treeNode (Floresta) e também capaz de importar Floresta a partir de um arquivo XML, o que será útil no software a ser embarcado. Aqui será tratada a parte de criação do arquivo XML. As funções criadas interessantes a este escopo são:

- **addNode:**
  - Cria um nó no arquivo XML com a marcação <node>, importando as informações de node, var, lowNode e highNode de um nó treenode.
- **plantArvoreXml:**

  - Cria uma árvore completa no arquivo XML com a marcação <arvore>, utilizando addNode. Copia todas as informações de uma arvore treeNode e atribuiu um número para a árvore com a marcação <numArvore>.

- **plantaFlorestaC:**
  - Cria um conjunto de árvores completo (Floresta) no arquivo XML com a marcação <floresta>, utilizando plantArvoreXml. Copia todas as informações de uma floresta treeNode.

A Figura 24 exibe o exemplo da criação de um arquivo XML com a função plantaFlorestaC. Este arquivo representa um sistema XOR (ou exclusivo), com uma única saída, ou seja, apenas 1 árvore. O sistema tem 3 nós mais os 2 nós terminais. A fins de comparação a Figura 25 exibe o armazenamento da mesma estrutura em um arquivo C, abordagem vista na última subseção.

Figura 24: Exemplo de arquivo XML gerado, sistema XOR.

```xml
<?xml version="1.0" encoding="utf-8"?>
<floresta>
    <arvore>
        <numArvore>0</numArvore>
        <node>
            <numNode>0</numNode>
            <var>-1</var>
            <lowNode>0</lowNode>
            <highNode>0</highNode>
        </node>
        <node>
            <numNode>1</numNode>
            <var>-1</var>
            <lowNode>1</lowNode>
            <highNode>1</highNode>
        </node>
        <node>
            <numNode>2</numNode>
            <var>0</var>
            <lowNode>3</lowNode>
            <highNode>4</highNode>
        </node>
        <node>
            <numNode>3</numNode>
            <var>1</var>
            <lowNode>0</lowNode>
            <highNode>1</highNode>
        </node>
        <node>
            <numNode>4</numNode>
            <var>1</var>
            <lowNode>1</lowNode>
            <highNode>0</highNode>
        </node>
    </arvore>
</floresta>
```

Fonte: Produzido pelo autor.

Figura 25: Exemplo de arquivo C gerado, sistema XOR.

```
#include "floresta.h"

treeNode arvoreC0[5] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,0,1},
 {4,1,1,0}
};
treeNode *florestaC[1] = {
arvoreC0};
```

Fonte: Produzido pelo autor.

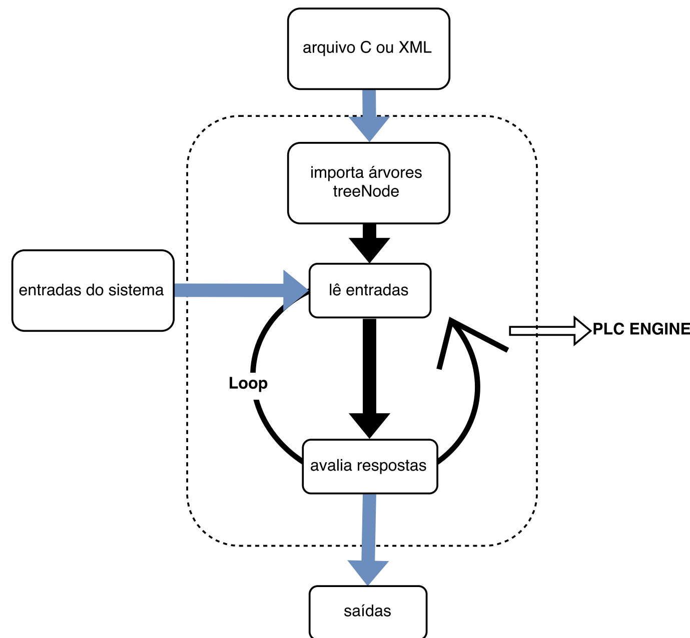
4.2 Programa PLC Engine

Na seção anterior foi mostrada a possibilidade de, a partir de um arquivo ESPRESSO, construir-se um conjunto de diagramas de decisão binária ordenados e reduzidos que representam as expressões lógicas de um sistema, relacionando suas entradas e saídas. Estes diagramas, representados por um conjunto de nós, formam árvores que por sua vez formam florestas. Estas florestas são armazenadas em matrizes treeNode em um arquivo C e também em um arquivo XML. Deve-se então elaborar um software, que será chamado de PLC Engine, ou apenas Engine, que seja capaz de absorver tais estruturas armazenadas e utilizá-las. Isto é, através de uma árvore BDD absorvida, dado um conjunto de entradas, Engine deve ser capaz de gerar um conjunto saídas binárias. Assim o software poderá avaliar todas as saídas do sistema e construir sua dinâmica de controle, ou seja, associar determinadas ações para cada saída, o que caracterizará seus estados.

A dinâmica do software PLC Engine é mostrada na Figura 26. Deseja-se importar as árvores treeNode de um dos arquivos exportados por Floresta (C ou XML). Após a importação entra-se em um loop infinito em que as entradas do sistema são lidas, as respostas são avaliadas de acordo com as entradas e as árvores treeNode que descrevem o BDD do sistema e, então, as saídas são configuradas. Lê-se novamente as entradas e repete-se o ciclo. Tais etapas do software são melhor descritas nas subseções a seguir.

Na Figura 27 pode-se ver o pseudocódigo de execução do software PLC Engine. Durante a execução, por uma única vez as árvores serão importadas de algum dos arquivos exportados por Floresta. Em seguida entra-se no loop eterno, onde a mesma sequência segue sendo executada. Nesta sequência lê-se os sinais de entrada, avalia-se as saídas do sistema, atualiza-se o estado do sistema e, entãom executa-se alguma ação externa, baseando-se nas saídas calculadas. No exemplo, há três saídas binárias e cada uma delas

Figura 26: Dinâmica do software PLC Engine.



Fonte: Produzido pelo autor.

é passada como estado (1 ou 0) para configuração de três saídas digitais. Nas subseções a seguir, explicar-se-á as funções de importação, atualização de saidas e atualização de estado desenvolvidas.

Figura 27: Pseudocódigo de execução de PLC Engine.

```
void configura()
{

    importaArvores();

}


void loop()
{

    while(1)
    {

        leEntradas();

        atualizaSaidas(florestaC);

        atualizaEstado();


        digitalWrite(0,saidas[0]);
        digitalWrite(1,saidas[1]);
        digitalWrite(2,saidas[2]);


    }

}
```

Fonte: Produzido pelo autor.

4.2.1 Importando Floresta a partir de um arquivo C.

Esta abordagem é bem simples e direta. Tendo o arquivo C declarado toda a floresta do sistema, basta incluí-lo e compilá-lo junto a Engine para se ter acesso direto às estruturas. A floresta estará contida na matriz de treeNode **florestac**. Cada elemento de florestaC será uma árvore do sistema, sendo que cada árvore representa uma saída BDD, como foi visto. Assim, caso se queira fazer referência à primeira árvore de um sistema, basta se fazer florestaC[0]. Se quiser-se fazer referência ao terceiro nó, por exemplo, desta árvore, faz-se: florestaC[0][2]. Como a matriz é do tipo treeNode, pode-se acessar as informações de cada nó diretamente. Na estrutura da Figura 25, por exemplo:

- florestaC[0][2].node = 2
- florestaC[0][2].var = 0
- florestaC[0][2].lowNode = 3
- florestaC[0][2].highNode = 4

4.2.2 Importando Floresta a partir de um arquivo XML

Aqui o processo de recuperação das informações dos diagramas BDD é um pouco mais extenso. Como já foi dito na subseção de exportação do arquivo XML, uma biblioteca extendida denominada **arvoreXML** foi criada a partir da biblioteca **Mini-XML**. Esta biblioteca além de conter as funções já citadas para exportação, contém também funções para importação e recuperação dos nós treeNode de uma floresta a partir de um arquivo XML. Estas funções são construídas a partir de diversas funções nativas da Mini-XML que não serão aqui discutidas, mas que são documentadas por Sweet (2003). Abaixo são descritas as funções desenvolvidas:

- **contaNodesXML:**
  - Conta a quantidade de nós (<nodes>) de uma árvore específica em um arquivo XML.
- **contaArvoresXML:**
  - Conta a quantidade de árvores (<arvores>) de um arquivo XML.
- **contaNodes2XML:**
  - Utiliza contaArvoresXML para descobrir quantas árvores existem em uma floresta de um arquivo XML e então utiliza contaNodesXML para contar a quantidade de nós que cada uma destas árvores têm.

- **alocaFloresta:**
  - Utiliza contaArvoresXML para saber quantas árvores existem em uma floresta de um arquivo XML. Em seguida utiliza contaNodes2XML para atribuir uma quantidade de nós para cada uma destas árvores. Feito isso a função aloca memória para estas árvores, formando uma matriz de nós treeNode a ser inicializada chamada **floresta_r**.
- **importNodeXML:**
  - Copia as informações (node, var, lowNode e highNode) de um nó de uma árvore específica de uma floresta descrita em um arquivo XML para um nó treeNode.
- **importArvoreXML:**
  - Utiliza importNodeXML para copiar as informações de todos os nós de uma árvore específica de uma floresta descrita em um arquivo XML para uma arvore treeNode.
- **importFlorestaXML:**
  - Utiliza importArvoreXML para copiar as informações de todos os nós, de todas as árvores de uma floresta descrita em um arquivo XML para uma floresta treeNode previamente alocada.

A função importFlorestaXML é então chamada e copia todas os valores de node, var, lowNode e highNode dos nós do arquivo XML para os nós treeNode alocados em floresta_r. A partir daí, pode-se fazer acesso direto às árvores e nós de árvores descritos no arquivo XML. Na estrutura do arquivo XML exibido anteriormente na Figura 24, por exemplo:

- floresta_r[0][2].node = 2
- floresta_r[0][2].var = 0
- floresta_r[0][2].lowNode = 3
- floresta_r[0][2].highNode = 4

4.2.3 Atualizando as saídas do sistema

Em posse da matriz treeNode representando os diagramas BDD do sistema, sejam eles importados de um arquivo .C ou .XML, pode-se finalmente utilizar as estruturas para avaliar o estado lógico de cada saída do sistema dados os estados lógicos de suas entradas.

Para isso é criada a função **int evaluateOutput2(treeNode *arvore, int *entradas)**. Esta função recebe como parâmetro a árvore **treeNode arvore** a ser avaliada e o vetor de entradas do sistema, entradas. Segue-se o seguinte procedimento:

1. Verifica em currentVar = arvore[nextBDD].var qual é a variável do nó raíz da árvore sendo avaliada. A variável nextBDD guarda sempre o próximo node a ser acessado.
2. Verifica em nextPath = entradas[currentVar] (vetor de entradas) qual é o estado lógico da variável var do passo anterior. Este estado lógico definirá qual caminho (0 - *low* ou 1 - *high*) tomar no próximo passo.
3. Se nextPath for high (estado lógico 1), configura-se o próximo nó como o filho de estado alto do nó atual, isto é, nextBDD = arvore[nextBDD].highNode. Caso nextPath seja low (estado lógico 0), configura-se o próximo nó como o filho de estado baixo do nó atual, nextBDD = arvore[nextBDD].lowNode.
4. Repete-se os passos anteriores até que o próximo nó seja um nó terminal, ou seja, que nextBDD seja 0 ou 1. Quando isto ocorre significa que chegou-se ao fim da árvore e que a resposta, ou seja, o estado lógico da saída é o próprio valor do nó terminal (0 ou 1). Este valor é então retornado pela função.

A função **atualizaSaidas** utiliza então evaluateOutput2 para definir o novo estado lógico de cada uma das saídas no vetor de saídas do sistema. Isto pode ser visto na Figura 28.

Figura 28: Função de atualização das saídas do sistema.

```
void atualizaSaidas(treeNode **floresta)
{
    int i;

    for (i = 0 ; i < nSaidas; i++)
    {
        saidas[i] = evaluateOutput2(floresta[i], entradas);
    }

}
```

Fonte: Produzido pelo autor.

4.2.4 Atualizando o estado do sistema

Como visto, na representação tabular de máquinas de estados os primeiros bits do vetor de entradas são reservados à identificação do estado corrente e os primeiros bits do vetor de saídas são reservados à identificação do próximo estado. Assim, tendo as novas saídas sido definidas, a função **atualizaEstado** atualiza no vetor de entradas estes primeiros valores reservados. Este procedimento pode ser visto na Figura 29.

Figura 29: Função de atualização do estado da máquina de estados.

```
void atualizaEstado()
{

    int i;

    for (i = 0 ; i < nEstados; i++)
    {
        entradas[i] = saidas[i];
    }
}
```

Fonte: Produzido pelo autor.

5 Resultados

Para demonstrar os resultados dos softwares gerados, acompanhar-se-á a resolução de um problema simples desde seu início, observando os resultados parciais de cada um dos programas gerados até o resultado final, que é a solução do problema em si, ou seja, o controle de uma planta simulada.

5.1 O Problema de Exemplo Proposto

Propõe-se, para teste, o problema simples do controle de portão de garagem. O problema consiste em controlar a posição de um portão de garagem com o auxílio de um botão. Caso o portão esteja totalmente fechado (parado na extrema direita) e o botão seja apertado, o portão deve se movimentar para a esquerda, caso contrário deve permanecer fechado. Caso o portão esteja totalmente aberto (parado na extrema esquerda) e o botão seja apertado ele deve se movimentar para a direita, caso contrário deve permanecer aberto. Se o portão estiver se movendo para a esquerda e chegar à extremidade esquerda (totalmente aberto), ele deve parar, caso contrário deve continuar se movendo para a esquerda. Se o portão estiver se movendo para a direita e chegar à extremidade direita (totalmente fechado), ele deve parar, caso contrário deve continuar se movendo para a direita. Caso o portão esteja se movendo em qualquer uma das direções e o botão seja apertado, o portão deve parar. Em um segundo acionamento do botão o portão deve voltar a se mover mas na direção contrária a que vinha se movendo.

Assim, este problema apresenta 6 estados:

- Estado 0 (000): Portão parado na direita (garagem totalmente fechada).
- Estado 1 (001): Portão em movimento para a esquerda.
- Estado 2 (010): Portão parado na esquerda (garagem totalmente aberta).
- Estado 3 (011): Portão parado entreaberto, vindo da esquerda.
- Estado 4 (100): Portão em movimento para a direita.
- Estado 5 (101): Portão parado entreaberto, vindo da direita.

Como o portão tem 6 estados, são necessários 3 bits para representá-los. Vimos que na representação tabular da máquina de estados os primeiros bits de entrada são utilizados para indicar o estado atual e os primeiros bits de saída são usados para indicar o próximo estado.

O sistema tem **3 entradas** binárias, relativas ao botão e aos sensores fim de curso. Com isso, teremos ao total 3+3 = 6 entradas binárias. Assim na representação tabular, teremos no vetor de entradas:

- **entrada 0 - Bit 0 (mais significativo) do estado atual do sistema.**
- **entrada 1 - Bit 1 do estado atual do sistema.**
- **entrada 2 - Bit 3 (menos significativo) do estado atual do sistema.**
- **entrada 3 - B**: Flag do botão, indica que o botão foi pressionado e ainda não foi atendido – 1 (botão requerido) e 0 (botão atendido ou não requerido).
- **entrada 4 - FCE**: Flag do sensor de fim de curso da esquerda, indica que o portão está em sua extremidade esquerda – 1 (verdadeiro) e 0 (falso).
- **entrada 5 - FCD**: Flag do sensor de fim de curso da direita, indica que o portão está em sua extremidade direita – 1 (verdadeiro) e 0 (falso).)

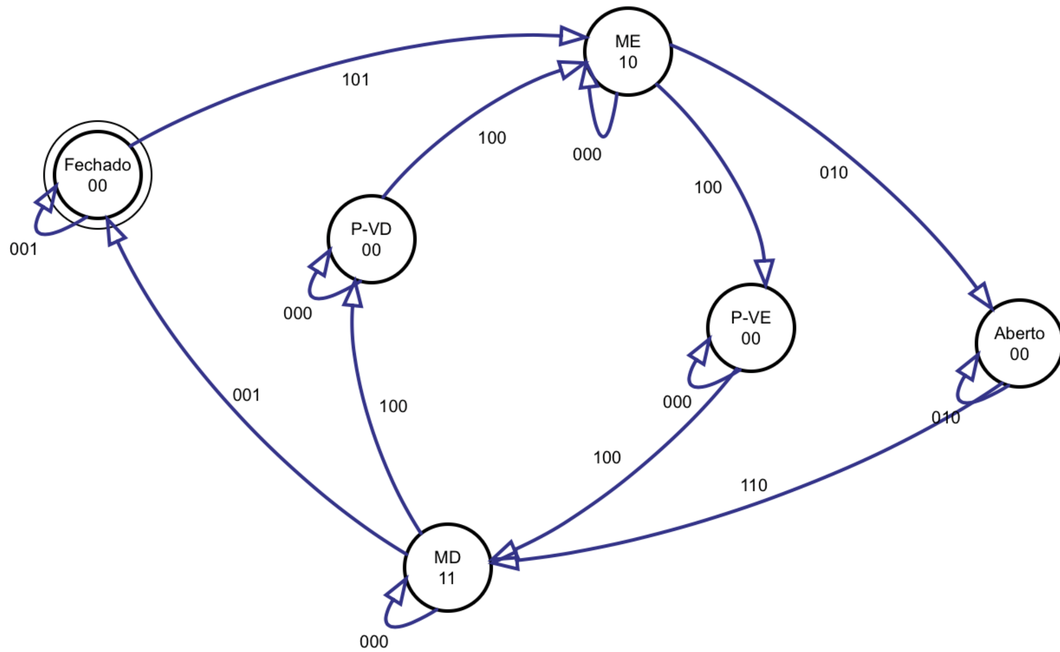
O sistema tem **2 saídas** binárias, relativas ao acionamento do motor e sua direção. Com isso, teremos ao total 2+3 = 5 saídas binárias. Assim na representação tabular, teremos no vetor de saídas:

- **saída 0 - Bit 0 (mais significativo) do próximo estado do sistema.**
- **saída 1 - Bit 1 do próximo estado do sistema.**
- **saída 3 - Bit 3 (menos significativo) do próximo estado do sistema.**
- **saída 4 - M (saída 0)**: Liga o motor, colocando o portão em movimento – (1 ligado) e 0 (desligado).
- **saída 5 - D (entrada 1)**: Seleciona a direção do movimento do portão – 1 (direita) e 0 (esquerda).

5.2 Resultados obtidos

Como vimos, para descrever a lógica do usuário, é utilizado o software Orange-Cad Web que permite a descrição gráfica do sistema e a geração do arquivo no formato ESPRESSO. O diagrama de estados projetado pode ser visto na Figura 30. O arquivo descritor resultante para o problema proposto pode ser visto na Figura 31. No diagrama de estados Fechado, ME, Aberto, P-VE, MD e P-VD referem-se aos estados 0, 1, 2, 3, 4 e 5, respectivamente.

Figura 30: Diagramas de estados do problema proposto.



Fonte: Produzido pelo autor.

Figura 31: Arquivo descritor ESPRESSO do problema proposto.

```
# =================================================
#   UFES - Universidade Federal do Espirito Santo
#   Descricao do circuito no formato ESPRESSO
#   Arquivo gerado automaticamente por OrangeCAD Web em
#   November 28, 2015 at 14:05:17 GMT-2
# =================================================
.i 6
.o 5
000101 00110
000001 00000
001010 01000
001100 01100
001000 00110
010110 10011
010010 01000
011100 10011
011000 01100
100001 00000
100100 10100
100000 10011
101100 00110
101000 10100
.e
```

Fonte: Produzido pelo autor.

5.2.1 Executando Floresta

O arquivo descritor exibido na Figura 31, será entrada para o programa **Floresta**, como vimos no capítulo 4. Como explicado, o software se valerá do arquivo para, primeiramente, gerar diagramas BDD para cada uma das 5 saídas do sistema. Estes diagramas formam tabelas BDD que são expostas na Figura 32. Vimos que em cada tabela a primeira coluna é o número de referência de cada nó, a segunda coluna é o número da entrada (ou variável – var) avaliada naquele nó. A terceira e a quarta coluna exibem o número de referência na tabela do próximo nó caso a variável avaliada no nó atual seja, respectivamente, 0 ou 1.

Figura 32: Diagramas BDD das saídas do problema proposto.

saída 0
```
ROOT: 85
[   13]   5:   1   0
[   24]   4:   0  13
[   25]   3:   0  24
[   38]   4:  13   0
[   39]   3:   0  38
[   43]   2:  25  39
[   44]   1:   0  43
[   65]   3:  38   0
[   83]   2:  38  65
[   84]   1:  83   0
[   85]   0:  44  84
```

saída 1
```
ROOT: 125
[   13]   5:   1   0
[   24]   4:   0  13
[   38]   4:  13   0
[   65]   3:  38   0
[   93]   3:  24   0
[  103]   3:  24  38
[  104]   2:   0 103
[  123]   2:  93  65
[  124]   1: 104 123
[  125]   0: 124   0
```

saída 2
```
ROOT: 152
[   12]   5:   0   1
[   13]   5:   1   0
[   38]   4:  13   0
[   39]   3:   0  38
[   65]   3:  38   0
[   80]   2:   0  65
[  129]   4:  12   0
[  130]   3:   0 129
[  139]   2: 130  38
[  142]   1: 139  80
[  150]   2:  39  38
[  151]   1: 150   0
[  152]   0: 142 151
```

saída 3
```
ROOT: 163
[   12]   5:   0   1
[   13]   5:   1   0
[   24]   4:   0  13
[   25]   3:   0  24
[   38]   4:  13   0
[   39]   3:   0  38
[   43]   2:  25  39
[   65]   3:  38   0
[  129]   4:  12   0
[  130]   3:   0 129
[  153]   2: 130  65
[  158]   1: 153  43
[  161]   2:  65  39
[  162]   1: 161   0
[  163]   0: 158 162
```

saída 4
```
ROOT: 164
[   13]   5:   1   0
[   24]   4:   0  13
[   25]   3:   0  24
[   38]   4:  13   0
[   39]   3:   0  38
[   43]   2:  25  39
[   44]   1:   0  43
[   65]   3:  38   0
[   66]   2:  65   0
[   67]   1:  66   0
[  164]   0:  44  67
```

Fonte: Produzido pelo autor.

Após gerar o BDD para cada saída, Floresta armazena os valores de node, var, lowNode e highNode em árvores treeNode chamadas "arvores". Em seguida, substitui os

valores de nodes destas árvores por seus próprios índices nas árvores, o que foi justificado também no capítulo 4. As árvores resultantes foram chamadas de arvoresIndexadas. O próximo passo é a exportação destas estruturas em 2 formatos: arquivo **estufa.c** (arquivo no formato C) e arquivo **floresta.xml** (arquivo no formato XML).

O arquivo estufa.c gerado é exibido na Figura 33.

Figura 33: Arquivo .C gerado pelo software Floresta na exportação das estruturas BDD para o problema proposto.

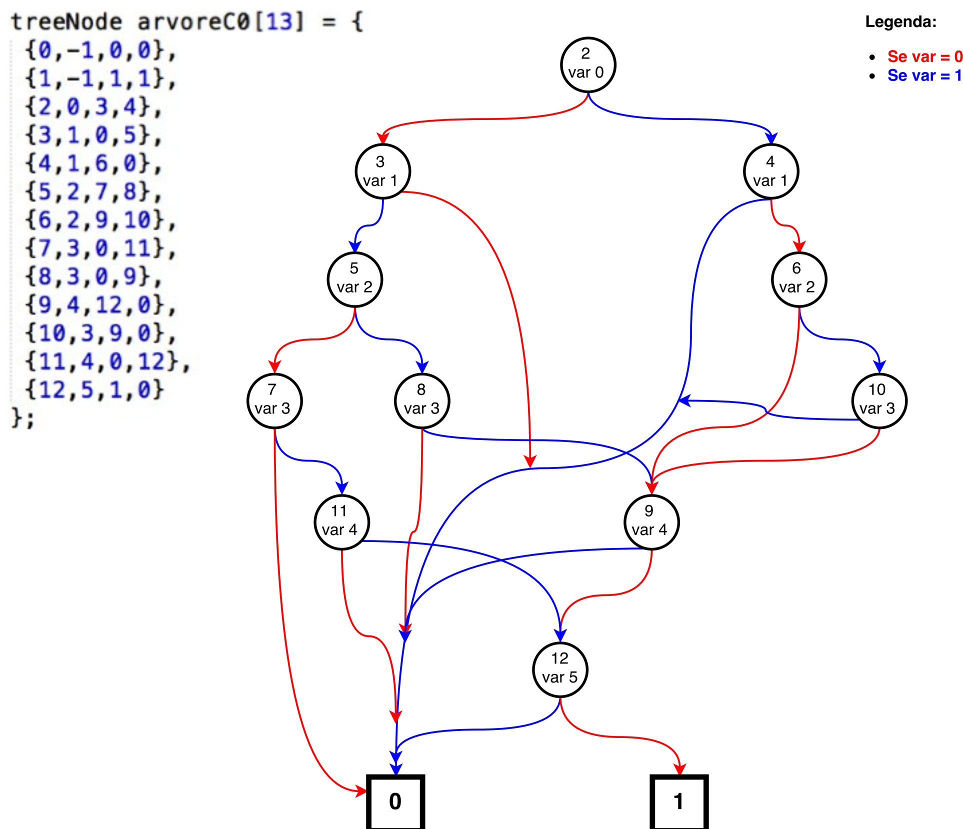
```
#include "floresta.h"

treeNode arvoreC0[13] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,0,5},
 {4,1,6,0},
 {5,2,7,8},
 {6,2,9,10},
 {7,3,0,11},
 {8,3,0,9},
 {9,4,12,0},
 {10,3,9,0},
 {11,4,0,12},
 {12,5,1,0}
};
treeNode arvoreC1[12] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,0},
 {3,1,4,5},
 {4,2,0,6},
 {5,2,7,8},
 {6,3,9,10},
 {7,3,9,0},
 {8,3,10,0},
 {9,4,0,11},
 {10,4,11,0},
 {11,5,1,0}
};
treeNode arvoreC2[15] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,5,6},
 {4,1,7,0},
 {5,2,8,9},
 {6,2,0,10},
 {7,2,11,9},
 {8,3,0,12},
 {9,4,13,0},
 {10,3,9,0},
 {11,3,0,9},
 {12,4,14,0},
 {13,5,1,0},
 {14,5,0,1}
};
treeNode arvoreC3[17] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,5,6},
 {4,1,7,0},
 {5,2,8,9},
 {6,2,10,11},
 {7,2,9,11},
 {8,3,0,12},
 {9,3,13,0},
 {10,3,0,14},
 {11,3,0,13},
 {12,4,15,0},
 {13,4,16,0},
 {14,4,0,16},
 {15,5,0,1},
 {16,5,1,0}
};
treeNode arvoreC4[13] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,0,5},
 {4,1,6,0},
 {5,2,7,8},
 {6,2,9,0},
 {7,3,0,10},
 {8,3,0,11},
 {9,3,11,0},
 {10,4,0,12},
 {11,4,12,0},
 {12,5,1,0}
};
treeNode *florestaC[5] = {
arvoreC0,
arvoreC1,
arvoreC2,
arvoreC3,
arvoreC4};
```

Fonte: Produzido pelo autor.

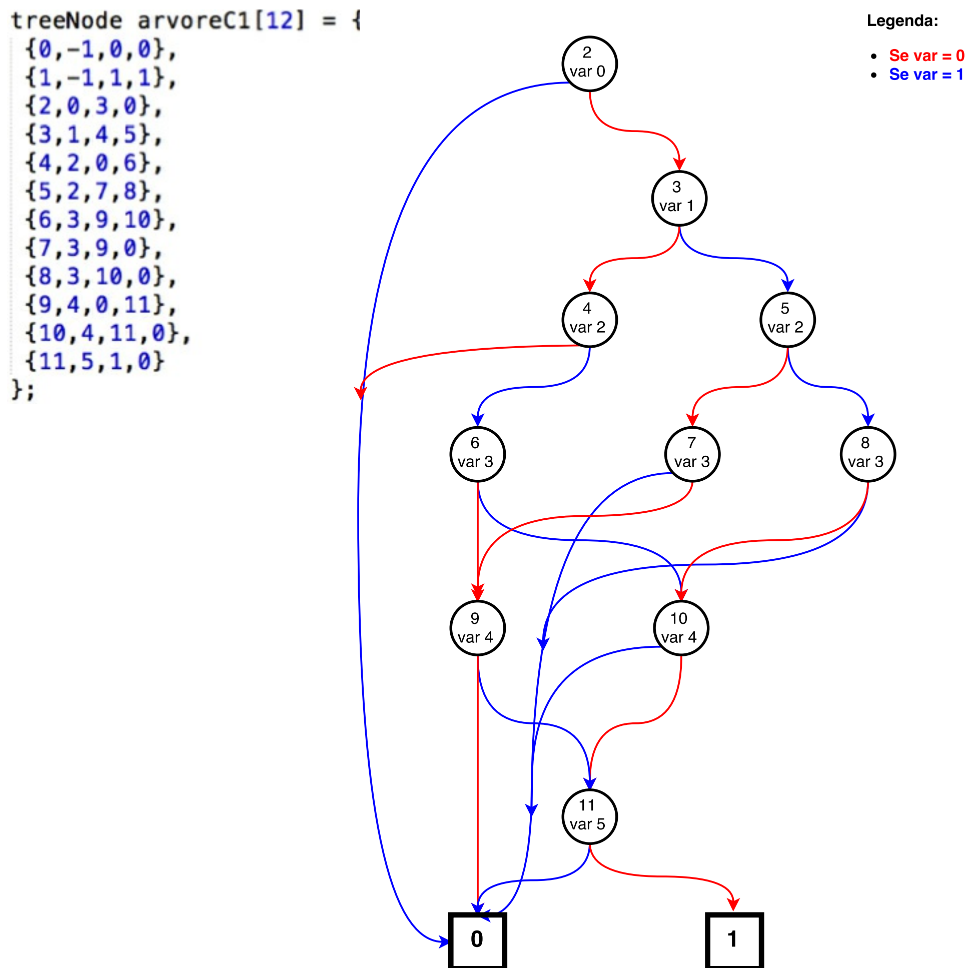
Pode-se ver que o arquivo C a ser incluído em PLC Engine contém uma estrutura florestaC com 5 árvores (arvoreC0, arvoreC1, arvoreC2, arvoreC3 e arvoreC4), uma para cada saída. O conteúdo de cada uma das árvores é exposto e, como se viu, cada treeNode das árvores armazena valores de node, var, lowNode e highNode, nesta ordem. A exceção é o campo var dos nós terminais 0 e 1. Como são nós terminais não há nenhuma variável (entrada) sendo avaliada nestes e então este campo é preenchido com -1 para sinalização. O terceiro treeNode de cada árvore é o nó raíz delas. Pode-se ver o gráfico em árvore das saídas 1, 2, 3, 4 e 5 na Figura 34, na Figura 35, na Figura 36, na Figura 37 e na Figura 38, respectivamente. Nestas figuras, parte-se do nó raíz de cada árvore, as linhas vermelhas sinalizam o próximo nó caso a variável de entrada daquele nó seja 0 e as linhas em azul indicam o próximo nó caso esta entrada seja 1. Os nós 0 e 1 contidos em quadrados simbolizam os nós terminais resultantes. Junto às arvores destaca-se cada uma das declarações correspondentes em estufa.c para melhor acompanhamento.

Figura 34: Gráfico em árvore do BDD da saída 1.



Fonte: Produzido pelo autor.

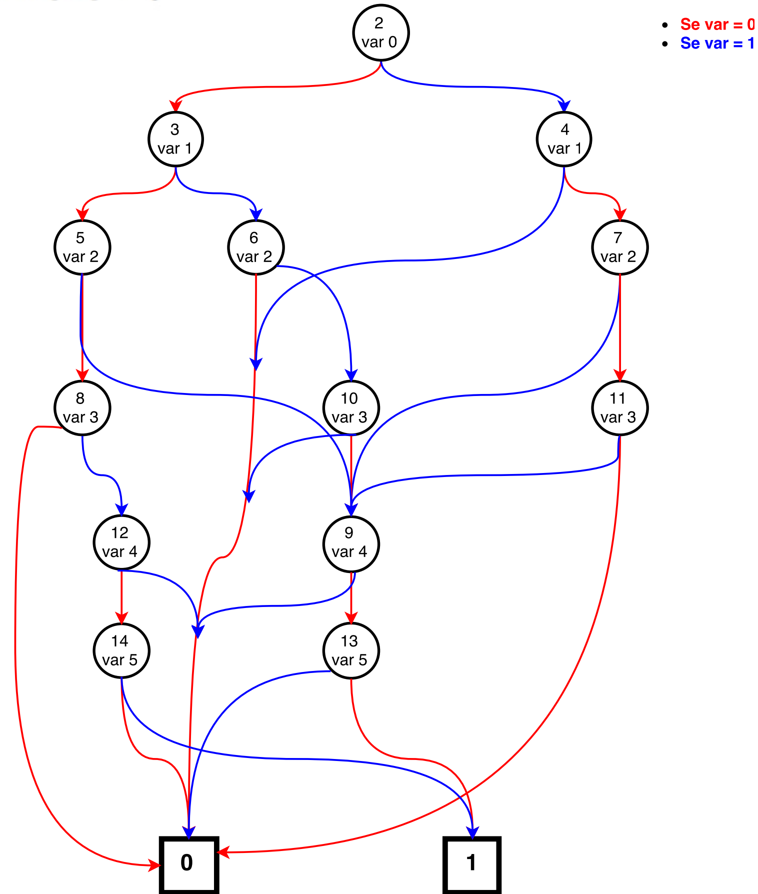
Figura 35: Gráfico em árvore do BDD da saída 2.



Fonte: Produzido pelo autor.

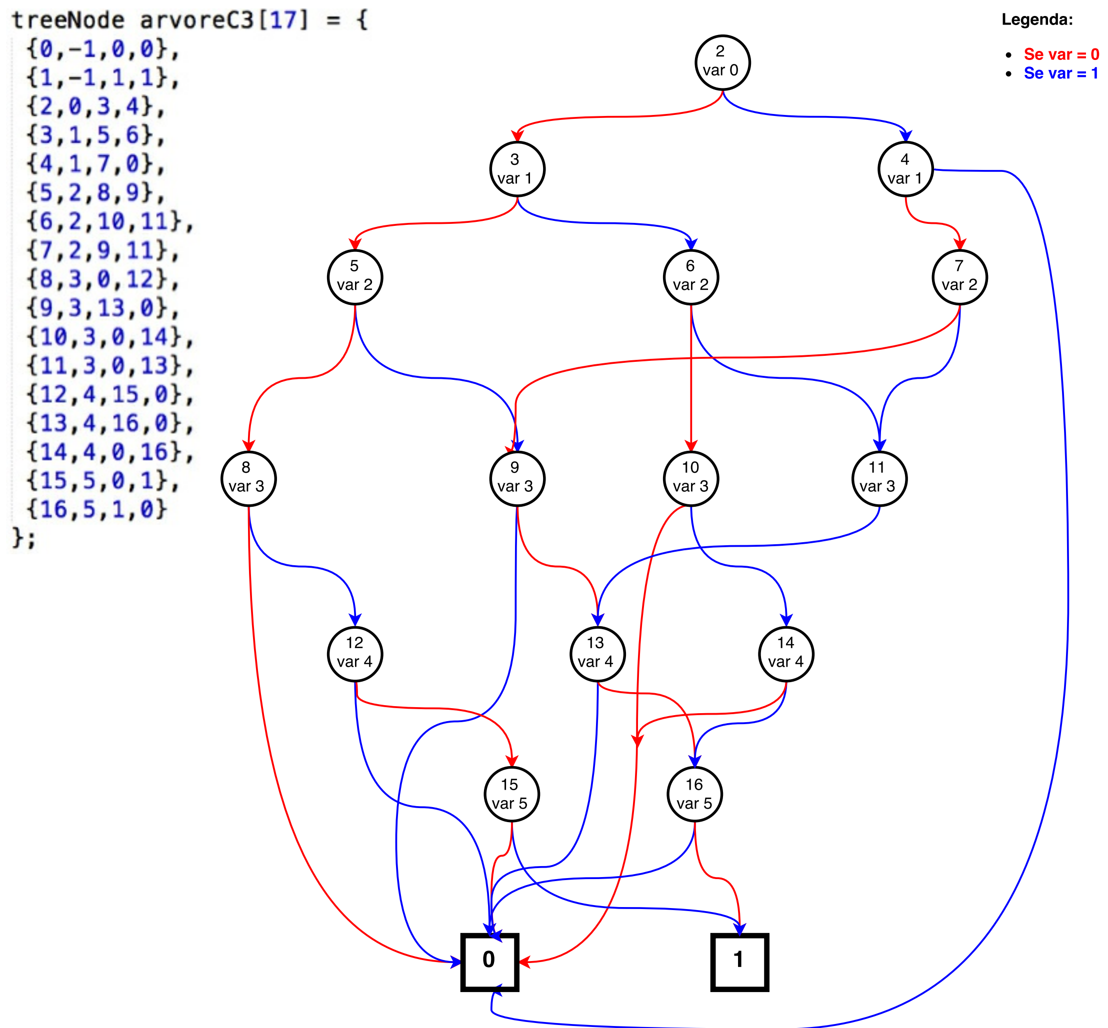
Figura 36: Gráfico em árvore do BDD da saída 3.

```
treeNode arvoreC2[15] = {
 {0,-1,0,0},
 {1,-1,1,1},
 {2,0,3,4},
 {3,1,5,6},
 {4,1,7,0},
 {5,2,8,9},
 {6,2,0,10},
 {7,2,11,9},
 {8,3,0,12},
 {9,4,13,0},
 {10,3,9,0},
 {11,3,0,9},
 {12,4,14,0},
 {13,5,1,0},
 {14,5,0,1}
};
```



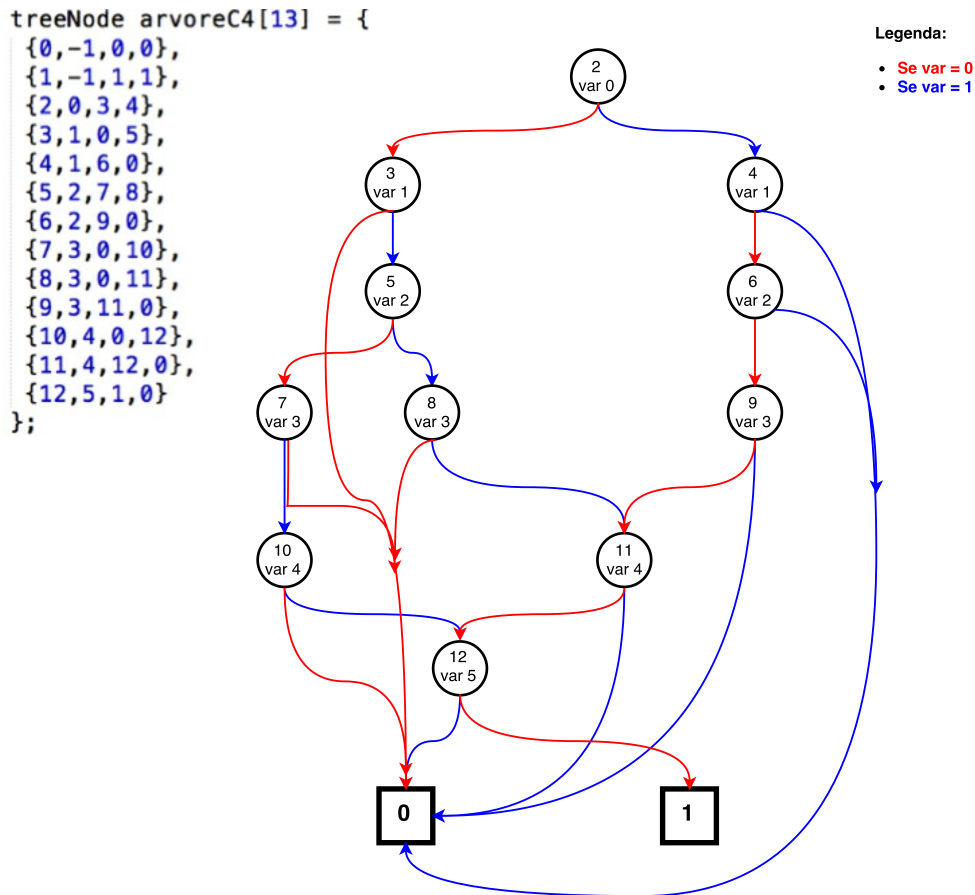
Fonte: Produzido pelo autor.

Figura 37: Gráfico em árvore do BDD da saída 4.



Fonte: Produzido pelo autor.

Figura 38: Gráfico em árvore do BDD da saída 5.



Fonte: Produzido pelo autor.

5.2.2 Executando PLC Engine

Em posse dos arquivos C e XML contendo as estruturas BDD geradas, pode-se iniciar a execução de PLC Engine. Primeiramente tais arquivos são importados. Apenas para este teste as árvores treeNode armazenadas são impressas e mostradas na Figura 39 para o arquivo C. Tem-se exatamente o mesmo resultado para o arquivo XML, que foi omitido. Note que a importação é feita com sucesso e os nós guardam exatamente os mesmos valores dos nós gerados em Floresta, que foram mostrados na Figura 33 para o arquivo C.

Figura 39: Árvores BDD importadas do arquivo .C ou XML pelo software PLC Engine

```
** Árvores importadas do arquivo .C **
Árvore 1:
node: 0, var: -1, lowNode: 0, highNode: 0
node: 1, var: -1, lowNode: 1, highNode: 1
node: 2, var: 0, lowNode: 3, highNode: 4
node: 3, var: 1, lowNode: 0, highNode: 5
node: 4, var: 1, lowNode: 6, highNode: 0
node: 5, var: 2, lowNode: 7, highNode: 8
node: 6, var: 2, lowNode: 9, highNode: 10
node: 7, var: 3, lowNode: 0, highNode: 11
node: 8, var: 3, lowNode: 0, highNode: 9
node: 9, var: 4, lowNode: 12, highNode: 0
node: 10, var: 3, lowNode: 9, highNode: 0
node: 11, var: 4, lowNode: 0, highNode: 12
Árvore 2:
node: 0, var: -1, lowNode: 0, highNode: 0
node: 1, var: -1, lowNode: 1, highNode: 1
node: 2, var: 0, lowNode: 3, highNode: 0
node: 3, var: 1, lowNode: 4, highNode: 5
node: 4, var: 2, lowNode: 0, highNode: 6
node: 5, var: 2, lowNode: 7, highNode: 8
node: 6, var: 3, lowNode: 9, highNode: 10
node: 7, var: 3, lowNode: 9, highNode: 0
node: 8, var: 3, lowNode: 10, highNode: 0
Árvore 3:
node: 0, var: -1, lowNode: 0, highNode: 0
node: 1, var: -1, lowNode: 1, highNode: 1
node: 2, var: 0, lowNode: 3, highNode: 4
node: 3, var: 1, lowNode: 5, highNode: 6
node: 4, var: 1, lowNode: 7, highNode: 0
node: 5, var: 2, lowNode: 8, highNode: 9
node: 6, var: 2, lowNode: 0, highNode: 10
node: 7, var: 2, lowNode: 11, highNode: 9
node: 8, var: 3, lowNode: 0, highNode: 12
node: 9, var: 4, lowNode: 13, highNode: 0
node: 10, var: 3, lowNode: 9, highNode: 0
node: 11, var: 3, lowNode: 0, highNode: 9
Árvore 4:
node: 0, var: -1, lowNode: 0, highNode: 0
node: 1, var: -1, lowNode: 1, highNode: 1
node: 2, var: 0, lowNode: 3, highNode: 4
node: 3, var: 1, lowNode: 5, highNode: 6
node: 4, var: 1, lowNode: 7, highNode: 0
node: 5, var: 2, lowNode: 8, highNode: 9
node: 6, var: 2, lowNode: 10, highNode: 11
node: 7, var: 2, lowNode: 9, highNode: 11
node: 8, var: 3, lowNode: 0, highNode: 12
node: 9, var: 3, lowNode: 13, highNode: 0
node: 10, var: 3, lowNode: 0, highNode: 14
node: 11, var: 3, lowNode: 0, highNode: 13
node: 12, var: 4, lowNode: 15, highNode: 0
node: 13, var: 4, lowNode: 16, highNode: 0
node: 14, var: 4, lowNode: 0, highNode: 16
node: 15, var: 5, lowNode: 0, highNode: 1
Árvore 5:
node: 0, var: -1, lowNode: 0, highNode: 0
node: 1, var: -1, lowNode: 1, highNode: 1
node: 2, var: 0, lowNode: 3, highNode: 4
node: 3, var: 1, lowNode: 0, highNode: 5
node: 4, var: 1, lowNode: 6, highNode: 0
node: 5, var: 2, lowNode: 7, highNode: 8
node: 6, var: 2, lowNode: 9, highNode: 0
node: 7, var: 3, lowNode: 0, highNode: 10
node: 8, var: 3, lowNode: 0, highNode: 11
node: 9, var: 3, lowNode: 11, highNode: 0
```

Fonte: Produzido pelo autor.

Para este teste algumas funções e bibliotecas foram adicionadas a PLC Engine para possibilitar a simulação. Destaca-se que estes recursos não são necessários quando o software for embarcado em alguma plataforma e cumprem aqui papel estritamente ligado à simulação. Entre as bibliotecas incluídas estão a de criação de *thread*, para criarmos uma *thread* que modificará as entradas do sistema e a de obtenção da hora local, para registrarmos o tempo durante a execução. Uma das funções criadas foi a denominada **keyHunter**. Esta rotina é responsável por monitorar o teclado do computador, que é utilizado para simular as entradas do sistema. As entradas 4, 5 e 6 (posições 3, 4 e 5 do vetor entradas) são ativadas pelas teclas **a**, **s** e **d** e simbolizam, como já descrito, o botão (B), o sensor de fim de curso esquerdo (FCE) e o sensor de fim de curso direito (FCD), respectivamente. As 3 entradas são alteradas durante a própria dinâmica de execução. O vetor de entradas é inicializado com [000001], portanto está no estado 000, isto significa que o portão inicia parado na extrema direita, ou seja, totalmente fechado. Significa ainda que a entrada do sistema físico é 001, portanto o botão está desativado (0), o fim de curso esquerdo está desativado (0) e o fim de curso direito está ativado (1).

A seguir, na Figura 40, na Figura 41 e na Figura 42, pode-se acompanhar o resultado do teste. Como também discutido durante o Capítulo 4, PLC Engine executa um *loop* eterno em que lê as entradas do sistema e atualiza o vetor de entradas, avalia as saídas com EvaluateOutput2, valendo-se das estruturas importadas, e atualiza o vetor de saídas. Em seguida, a *engine* atualiza o estado do sistema, utilizando os valores do vetor

de saídas que identificam o próximo estado para alimentar parte do vetor de entradas responsável por armazenar o estado atual. Durante a execução, as entradas podem ser alteradas pelo teclado, como dito. Ao final de cada *loop* todos os valores das entradas e das saídas são impressos, bem como o *status* do portão. O *status* identifica cada um dos 5 estados enunciados, podendo estar: **Parado e fechado...**, **movendo para a esquerda**, **parado entreaberto...**, **movendo para a direita** e **parado e aberto...**.

Figura 40: Simulação de PLC Engine para o problema proposto, parte I.

```
            Entradas
B[0] FCE[0] FCD[1] || Estado atual:[000]

            Saidas
M[0] D[0] || Novo estado:[000]

Status do portao: parado e fechado...          1
Tempo: 17:14:54
--------------------------------------------
            Entradas
B[1] FCE[0] FCD[1] || Estado atual:[000]

            Saidas
M[1] D[0] || Novo estado:[001]                 2

Status do portao: movendo para a esquerda <---
Tempo: 17:15:32
--------------------------------------------
            Entradas
B[0] FCE[0] FCD[0] || Estado atual:[001]

            Saidas
M[1] D[0] || Novo estado:[001]                 3

Status do portao: movendo para a esquerda <---
Tempo: 17:15:34
d
--------------------------------------------
            Entradas
B[1] FCE[0] FCD[0] || Estado atual:[001]

            Saidas
M[0] D[0] || Novo estado:[011]                 4

Status do portao: parado entreaberto...
Tempo: 17:15:35
--------------------------------------------
```

```
            Entradas
B[0] FCE[0] FCD[0] || Estado atual:[011]

            Saidas
M[0] D[0] || Novo estado:[011]

Status do portao: parado entreaberto...        5
Tempo: 17:15:38
--------------------------------------------
            Entradas
B[1] FCE[0] FCD[0] || Estado atual:[011]

            Saidas
M[1] D[1] || Novo estado:[100]                 6

Status do portao: movendo para a direita --->
Tempo: 17:15:39
--------------------------------------------
            Entradas
B[0] FCE[0] FCD[0] || Estado atual:[100]

            Saidas
M[1] D[1] || Novo estado:[100]                 7

Status do portao: movendo para a direita --->
Tempo: 17:15:43

--------------------------------------------
            Entradas
B[1] FCE[0] FCD[0] || Estado atual:[100]

            Saidas
M[0] D[0] || Novo estado:[101]                 8

Status do portao: parado entreaberto...
Tempo: 17:15:43
--------------------------------------------
```

Fonte: Produzido pelo autor.

Na Figura 40, em 1, vemos o início da execução em 17:14:54, com o portão parado e fechado. O vetor de entradas foi inicializado com [000001], portanto o estado atual vale [000] e as entradas físicas valem B[0], FCE[0] e FCD[1]. O vetor de saídas está preenchido com [00000], portanto o próximo estado calculado ainda é [000] e as saídas físicas são M[0] e D[0], assim o portão está com status parado e fechado, o que era exatamente o esperado. Em 2 (17:15:32), o botão é pressionado (B[1]), a saída muda então para [10], ou seja, com o motor movendo (M[1]) na direção 0 (D[0]). Com isso o portão começa a se mover para a esquerda, como o status mostra. O novo estado calculado é [001]. Em 3 (17:15:34) o portão continua se movendo para a esquerda como esperado, já que nem o botão foi pressionado nem o fim de curso esquerdo foi ativado. Em 4 (17:15:35), o botão é pressionado novamente o que faz a saída ir para [00], ou seja, com o portão sem movimento (M[0]). O status então é parado entreaberto. Em 5 (17:15:38) o portão ainda está parado

já que o botão ainda não foi pressionado B[0]. Em 6 (17:15:39), isto muda e o botão é pressionado B[1], a saída vai para [11], ou seja, com o portão se movendo para a direita, como indica o status, o que é correto, já que antes o portão se movia para a direita. Em 7 (17:15:43) o portão ainda está se movendo para a direita, quando menos de 1 segundo depois, em 8 (17:15:43), o botão é novamente pressionado fazendo a saída ir para [00], o que faz com que o portão pare, pois tem-se M[0].

Figura 41: Simulação de PLC Engine para o problema proposto, parte II.

```
             Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[101]

               Saidas
 M[0] D[0] || Novo estado:[101]

Status do portao: parado entreaberto...   9
Tempo: 17:15:45

--------------------------------------------
             Entradas
 B[1] FCE[0] FCD[0] || Estado atual:[101]

               Saidas                      10
 M[1] D[0] || Novo estado:[001]

Status do portao: movendo para a esquerda <---
Tempo: 17:15:46
--------------------------------------------
             Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[001]

               Saidas                      11
 M[1] D[0] || Novo estado:[001]

Status do portao: movendo para a esquerda <---
Tempo: 17:15:49
a
--------------------------------------------
             Entradas
 B[0] FCE[1] FCD[0] || Estado atual:[001]

               Saidas                      12
 M[0] D[0] || Novo estado:[010]

Status do portao: parado e aberto...
Tempo: 17:15:49
--------------------------------------------
```

```
             Entradas
 B[0] FCE[1] FCD[0] || Estado atual:[010]

               Saidas
 M[0] D[0] || Novo estado:[010]

Status do portao: parado e aberto...           13
Tempo: 17:15:55
d
--------------------------------------------
             Entradas
 B[1] FCE[1] FCD[0] || Estado atual:[010]

               Saidas                          14
 M[1] D[1] || Novo estado:[100]

Status do portao: movendo para a direita --->
Tempo: 17:15:55
--------------------------------------------
             Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[100]

               Saidas                          15
 M[1] D[1] || Novo estado:[100]

Status do portao: movendo para a direita --->
Tempo: 17:15:59
d
--------------------------------------------
             Entradas
 B[1] FCE[0] FCD[0] || Estado atual:[100]

               Saidas                          16
 M[0] D[0] || Novo estado:[101]

Status do portao: parado entreaberto...
Tempo: 17:16:00
--------------------------------------------
```

Fonte: Produzido pelo autor.

Em sequência, na Figura 41, vê-se que em 9 (17:15:45) o portão permanece parado, pois o botão ainda não foi pressionado. Quase no mesmo instante, em 10 (17:15:46), o botão é pressionado (B[1]) e a saída vai para [10], com o portão se movendo para a esquerda. Mais uma vez a mudança de direção é correta, pois o portão anteriormente estava se movendo para a direita (estado 101). Em 11 (17:15:49), como esperado, o portão continua se movendo para a esquerda, já que nem o botão foi pressionado nem o fim de curso esquerdo foi ativado. Logo após, em 12 (17:15:49), o fim de curso esquerdo é ativado (FCE[1]) e o portão para, o que significa que o portão chegou na extremidade esquerda. Assim seu status é indicado corretamente: parado e aberto. Em 13 (17:15:55), o portão ainda está parado e aberto pois o botão ainda não foi pressionado (B[0]). O botão é pressionado em 14 (17:15:55), fazendo o portão mudar corretamente de direção, movendo-

se para a direita (saídas em [11]). Em 15 (17:15:49), o botão ainda não foi pressionado nem o fim de curso direito foi ativado e então, corretamente, o portão permanece em movimento para a direita (saídas em [11]), pode-se ver que o estado atual e o novo estado são o mesmo. Depois, em 16 (17:16:00), o botão é pressionado (B[1]) e o portão, corretamente, fica parado entreaberto.

Figura 42: Simulação de PLC Engine para o problema proposto, parte III.

```
           Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[101]

           Saidas
 M[0] D[0] || Novo estado:[101]
                                              17
Status do portao: parado entreaberto...
Tempo: 17:16:02
d
--------------------------------------------
           Entradas
 B[1] FCE[0] FCD[0] || Estado atual:[101]

           Saidas                             18
 M[1] D[0] || Novo estado:[001]

Status do portao: movendo para a esquerda <---
Tempo: 17:16:03
--------------------------------------------
           Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[001]

           Saidas                             19
 M[1] D[0] || Novo estado:[001]

Status do portao: movendo para a esquerda <---
Tempo: 17:16:05
d
--------------------------------------------
           Entradas
 B[1] FCE[0] FCD[0] || Estado atual:[001]

           Saidas                             20
 M[0] D[0] || Novo estado:[011]

Status do portao: parado entreaberto...
Tempo: 17:16:06
--------------------------------------------
```

```
           Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[011]

           Saidas
 M[0] D[0] || Novo estado:[011]
                                              21
Status do portao: parado entreaberto...
Tempo: 17:16:08
d
--------------------------------------------
           Entradas
 B[1] FCE[0] FCD[0] || Estado atual:[011]

           Saidas                             22
 M[1] D[1] || Novo estado:[100]

Status do portao: movendo para a direita --->
Tempo: 17:16:08
--------------------------------------------
           Entradas
 B[0] FCE[0] FCD[0] || Estado atual:[100]

           Saidas                             23
 M[1] D[1] || Novo estado:[100]

Status do portao: movendo para a direita --->
Tempo: 17:16:12

--------------------------------------------
           Entradas
 B[0] FCE[0] FCD[1] || Estado atual:[100]

           Saidas                             24
 M[0] D[0] || Novo estado:[000]

Status do portao: parado e fechado...
Tempo: 17:16:12
--------------------------------------------
```

Fonte: Produzido pelo autor.

Em seguida, em 17 (17:16:02), o portão continua parado, pois o botão ainda não foi pressionado (B[0]). Quase no mesmo instante, em 18 (17:16:03), o portão começa a se mover para a esquerda, pois o botão é pressionado (B[1]) e em seu último movimento ele estava se movendo para a direita (estado 101). O portão continua se movendo para a esquerda em 19 (17:16:05), pois tanto o botão quanto o fim de curso esquerdo ainda estão em estado baixo. Em 20 (17:16:06), o botão é ativado (B[1]) e o portão para (saídas em [00]). Segue parado em 21 (17:16:08), pois o botão ainda não foi ativado. Em 22 (17:16:08), o portão começa a se mover para a direita, pois o botão é pressionado (B[1]) e em seu último movimento ele estava se movendo para a esquerda (estado 011). Segue movendo para a direita em 23 (17:16:12), pois nem o botão foi pressionado (B[0]), nem o fim de curso direito foi ativado ainda (FCD[0]). Por fim o fim de curso direito é ativado em 24 (17:16:12) e o portão exibe status parado e fechado (saídas em [00]), voltando ao estado

inicial [000]. Assim o portão se comportou como esperado e tanto suas saídas físicas quanto suas transições de estado foram regidas pelas descrição lógica passada, implementando corretamente a máquina de estados proposta.

6 Conclusão

A proposta principal do trabalho era a criação de dois software que, juntos, fossem capazes de representar automaticamente uma máquina de estados finitos, a partir da descrição do usuário, para servir de base para a implementação de software de um PLC. O primeiro programa é executado em ambiente de desenvolvimento, em um computador genérico, e o segundo é executado em um sistema embarcado.

Viu-se que o primeiro programa, denominado Floresta, é capaz de absorver tal descrição e gerar automaticamente, sem a necessidade de praticamente nenhuma interação com o usuário, uma representação canônica e mínima deste sistema. Tal representação é possível utilizando-se diagramas de decisão binária, técnica bastante referenciada pela literatura. Floresta, em seguida, cria uma estrutura simples para copiar os BDD do sistema, para que assim, o segundo programa, a ser embarcado, não dependa de nenhuma biblioteca de terceiros para fazer uso dos diagramas.

O primeiro programa é ainda capaz de exportar os BDD gerados em dois formatos. O primeiro formato, um simples arquivo C, é facilmente incluído em qualquer programa escrito em C e o acesso às suas estruturas de dados é direto. O segundo formato é um arquivo XML, que apesar de mais complexo e de exigir maior esforço na exportação e na importação, traz a vantagem de ser intercambiável entre programas de linguagens diferentes. Isso pode ser útil inclusive para aplicações futuras. Isto é, se o usuário preferir, por exemplo, utilizar apenas o primeiro programa desenvolvido, pode utilizar o arquivo XML gerado e desenvolver um software em qualquer outra linguagem que absorva a lógica ali descrita.

O segundo programa, definido como PLC Engine, é, como o nome diz, o motor da execução da máquina de estados. É capaz de absorver, com mínima interação com o usuário, os dois tipos de arquivos gerados. Destaca-se que a absorção do arquivo C, como dito, é muito simples e direta, mas é firmado o compromisso de se ter que realizar a importação em tempo de compilação. A absorção do arquivo XML é um pouco mais elaborada mas traz a vantagem de poder ser feita em tempo de execução. Isso é, o usuário pode escrever rotinas que alterem, em tempo de execução, a máquina de estados finitos utilizada, o que traz grande flexibilidade e poder de adaptação ao sistema. PLC Engine pode então, depois de ler as entradas e o estado corrente do sistema, calcular todas as suas saídas, bem como o seu próximo estado, percorrendo os diagramas de decisão binária armazenados. Isso faz com que, juntos, os programas sejam capazes de representar máquinas de estados finitos de forma eficiente e com baixa intervenção do usuário.

Desta forma, destaca-se que o conjunto de software desenvolvido pode, de fato,

servir de base para a implementação de software de um PLC. Para tal seria necessário ainda o desenvolvimento de algumas ferramentas auxiliares, principalmente na interface com usuário. Como visto, neste trabalho, o usuário descreve o sistema a ser controlado definindo um diagrama de estados em ambiente gráfico, e tal ambiente gera como saída um arquivo do tipo ESPRESSO. Entretanto, os diagramas de estados não são a forma habitual que operadores de PLC utilizam para essa realizar essa descrição. Para o refinamento dessa solução, seria necessário o desenvolvimento de ferramentas que permitissem que o usuário descrevesse o sistema em linguagens de descrição comumente usadas em PLCs, como *Ladder*, e em seguida gerassem o respectivo arquivo ESPRESSO. Feito isso, Floresta poderia, da mesma forma, gerar os arquivos C e XML representando de forma eficiente o sistema. Tais arquivos poderiam então, junto das funções implementadas em PLC Engine, ser embarcados em um microcontrolador.

Seria também necessário, idealmente, o embarque de um sistema operacional de tempo real no microcontrolador, que pudesse garantir, entre outras coisas, taxas de amostragem fixas para a leitura das entradas do sistema e o correto escalonamento de execução das funções desenvolvidas.

Ressalta-se que o desenvolvimento de hardware para um PLC baseado em microcontroladores, foge ao escopo deste trabalho e é um problema complexo, o qual deve seguir diretrizes de segurança, com vários níveis de redundância. Mas, acredita-se que, uma vez desenvolvido tal hardware, o trabalho aqui apresentado pode contribuir para o desenvolvimento do software a ser nele embarcado, o que satisfaz a motivação inicial e as pretensões do projeto.

Referências

AKERS, S. B. Binary decision diagrams. *Computers, IEEE Transactions on*, IEEE, v. 100, n. 6, p. 509–516, 1978. Citado na página 36.

ANDERSEN, H. R. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen*, 1997. Citado na página 37.

BLIESENER, R.; EBEL, F.; LÖFFLER, C. *Programmable Logic Controllers Basic Level*. [S.l.]: Festo Didactic Co, 2002. Citado na página 24.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, IEEE, v. 100, n. 8, p. 677–691, 1986. Citado na página 37.

FLORES, P. *Utilização de Diagramas de Decisão Binária (BDDs) para Representar Máquinas de Estado (FSMs) e Verificar Fórmulas de Lógica Temporal (CTL)*. [S.l.], 1996. Disponível em: <http://algos.inesc.pt/~pff/home/publications/Flores-INESC96-ctl.pdf>. Citado 4 vezes nas páginas 35, 36, 37 e 38.

GILL, A. et al. Introduction to the theory of finite-state machines. McGraw-Hill, 1962. Citado na página 29.

KNUTH, D. E. A draft of section 7.1. 4: Binary decision diagrams. *The Art of Computer Programming*, v. 4A, 2008. Citado na página 35.

KNUTH, D. E. *The art of computer programming. Vol. 4A. , Combinatorial algorithms. Part 1*. Upple Saddle River (N.J.), London, Paris: Addison-Wesley, 2011. La couv. porte en plus : The classic work extented and refined. ISBN 978-0-201-03804-0. Disponível em: <http://opac.inria.fr/record=b1132643>. Citado na página 35.

LEE, C.-Y. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, Wiley Online Library, v. 38, n. 4, p. 985–999, 1959. Citado na página 36.

LIND-NIELSEN. *Buddy: A BDD package*. [S.l.], 2007. Disponível em: <http://buddy.sourceforge.net/manual/main.html>. Citado 2 vezes nas páginas 46 e 47.

LIND-NIELSEN, J. et al. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, Springer, v. 18, n. 1, p. 5–23, 2001. Citado na página 46.

PTOLEMAEUS, C. (Ed.). *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. Disponível em: <http://ptolemy.org/books/Systems>. Citado na página 29.

RIBEIRO, G. S. *OrangeCad Web*: Uma ferramenta de apoio à síntese de circuitos lógicos sequenciais. Dissertação (Trabalho de Conclusão de Curso) — Universidade Federal do Espírito Santo, 2011. Citado na página 44.

SAMEK, P. U. M. Statecharts in c/c++: Event-driven programming for embedded systems. *Newnes, Newton, MA*, 2008. Citado na página 29.

SWEET, M. R. *Mini-XML*: Tiny xml library. [S.l.], 2003. Disponível em: <http://www.msweet.org/projects.php?Z3>. Citado 2 vezes nas páginas 57 e 61.

WIESEN, H. B. G. *What Is a Software Engine?* 2015. Disponível em: <http://www.wisegeek.com/what-is-a-software-engine.htm>. Acesso em: 03 jun 2015. Citado na página 26.